静的解析による並列論理型言語KL1の実行最適化に関する研究

大野 和彦

1998年1月

静的解析による並列論理型言語KL1の実行最適化に関する研究

大野　和彦

1998年1月

概要

並列論理型言語 KL1 は、動的なデータ構造を容易に扱うことができることや、暗黙の通信・同期により並列化によるバグが混入しにくいことなど、様々な利点を持ち、記号処理や知識処理といった非数値分野の並列プログラム記述に適している。しかしその反面、現在までの KL1 の実装は、C などの手続き型言語の実装に比べ、実行効率が劣るという問題がある。そこで本研究では、静的解析情報を利用した実行最適化によって、KL1 処理系の実装方式の改良を行った。

KL1 では、実行時に決定する要因が多く、それらを動的処理するオーバヘッドが速度低下を生じさせている。これを解決するには、プログラムを静的解析し、その結果を利用してプログラムを最適化する手法が有効である。解析によってプログラムの振る舞いの一部が決定できれば、その情報を用いて動的処理の一部を静的処理したコードを生成でき、実行時のオーバヘッドを削減することができる。

本研究では、まず様々な最適化に応用可能な KL1 プログラムの静的解析手法として、制約充足による型解析を提案した。本手法では型集合や型変数の導入によって、プログラムの非決定的な部分や再帰構造を伴う部分が、効率よく表現される。また、必要に応じて型情報を拡張することで、様々な用途に対応できる。さらに、制約充足による解析を行うことで、双方向のデータフローの存在やゴールの実行順の動的変化といった、KL1 プログラム解析上の問題点を解決している。

次に、この解析手法を用いて、2 種類の実行最適化手法を提案した。

まず、ノード間通信の最適化手法として、選択的一括送信手法を提案した。一般に KL1 のプログラムでは、各データが実際に参照されるまではそれが必要になるかどうかが判明しない。そのため従来の処理系では、他ノード上のデータを参照する場合には、個々の要素について要求駆動的なデータのやりとりを行う。この結果、細粒度の通信が発生し、そのオーバヘッドが速度低下につながる。

これに対し選択的一括送信手法では、静的解析により各並列プロセスで参照されるデータ要素を調べ、必要な要素をまとめ送りする選択的一括送信コードを生成する。そして、実行時のデータ要求に対し、対応する送信コードを実行することによって、余分なデータ転送を生じずに通信回数を削減する。

この手法を KL1 処理系 KLIC に実装し性能評価を行った結果、総転送量を増加させることなく、通信回数を従来方式の 2/3～1/200 程度と大幅に削減でき、多くの場合で 2～5 倍程度の速度向上が得られた。

また、ゴール・スケジューリングの最適化手法として、ゴールのスレッド化手法を提案した。従来の方式では、ゴールを並行制御の単位とするため細粒度の動的スケジューリングが行われ、そのオーバヘッドが大きなものになる。スレッド化手法では、静的解析によりゴール間の依存を調べ、逐次化可能なゴール群を 1 スレッドとして静的スケジューリングする。これによって、スレッド内ゴールの動的スケジューリング・オーバヘッドを削減することができる。また、スレッドを動的スケジューリングすることで、プログラムが持つ本質的な並行性が保たれる。さらに、スレッ

ド内ゴールをスタックで管理することにより、ヒープの消費量を減らし、ガーベジコレクションの回数を削減することができる。

この手法を用いることで、並行性の粒度が上がる反面、並列性が低下し、一部のノードのアイドル時間が増加して実行速度が低下するという問題が生じる。本研究ではこの問題を解決するために、スレッドの reply first スケジューリング方式を提案した。この方式では、他ノードからの値要求メッセージに対し、その値を生成するスレッドを優先的にスケジューリングする。これによって、値要求の受信から返信までの時間が最短化されるため、アイドル時間を短縮し、並列性を向上させることができる。

これらの手法を KLIC に実装し性能評価を行った結果、逐次実行において、従来方式ではオーバヘッドが大きいプログラムで数倍、小さいものでも 1.3 倍程度の速度向上が得られ、動的スケジューリングのオーバヘッドを大きく削減できることが確認できた。またガーベジコレクション回数も、従来方式の 1/2〜1/7 と大幅に削減された。さらに、reply first 方式を用いることにより、並列実行でも並列性低下の問題を改善でき、逐次の場合と同程度の性能向上が得られた。

以上の結果より、本研究で提案した静的解析手法は、適切な拡張を加えることで、様々な静的情報を得る手段として活用できることが示された。また、これを利用した 2 種類の最適化手法は、性能評価によりいずれの場合も高い効果が得られ、KL1 処理系の実行最適化手法として有効であることが確認できた。

目 次

図目次

表目次

Chapter 1

緒論

1.1 研究の目的

近年、並列計算機の発達に伴い、並列処理プログラムを記述するための様々な言語の研究が行われている。第五世代コンピュータプロジェクトの核言語である並列論理型言語 KL1 [11] はその中の一つであり、動的なデータ構造を容易に扱うことができることや、暗黙の通信・同期により並列化によるバグが混入しにくいことなど、様々な利点を持つ。そのため KL1 は、とくに記号処理や知識処理といった非数値分野の並列プログラム記述に適しており、従来より多種の応用プログラムが作成されてきた [12, 13]。また、最近では KLIC [14] と呼ばれる移植性に優れた処理系が開発され、様々なワークステーションや並列計算機上で利用可能になっている。

しかしながら、現在までの KL1 の実装は、C や FORTRAN のような手続き型言語をベースとした並列言語の実装に比べると、速度やメモリ効率が劣るという問題がある。このため、実用的なプログラムの作成には、実装方式を改良し、より効率的なものにする必要がある。

本研究の目的は、この KL1 処理系の実装方式を改良することである。KL1 は、Prolog を始めとする他の論理型言語の多くと同様に、メモリの確保・解放は自動的に行われ、データ型の宣言も不要である。また、通信や同期といった並行・並列性の制御も、プログラム中で明示的に記述する必要がない。これらのことは、並列プログラムの作成を容易にする反面、実行系に大きな負担をもたらしている。すなわち、動的な判断や操作が手続き型言語に比べて多く、その処理のオーバヘッドが効率を大きく低下させる。

これを解決するには、プログラムを静的解析し、その結果を利用してプログラムを最適化する手法が有効である。解析によってプログラムの振る舞いを決定できれば、その情報を用いて最適化されたコードを生成することができる。この結果、従来は動的に処理していた部分を減らし、実行時のオーバヘッドを削減することができる。

1.2 研究の成果

本研究は、KL1 プログラムを静的解析に基づき効率良く実行する方式を見出すことを目的として行い、以下の三つの成果を得た。

1. 制約充足による型解析の提案

2. ノード間通信の最適化

3. ゴール・スケジューリングの最適化

1.2.1 制約充足による型解析の提案

論理型言語の静的解析手法としては、従来より様々な方法が提案されてきているが、本研究では、実際にプログラム中に出現する項に対応した構造型データ型と型変数を導入した。これによって、再帰構造を含めたデータ構造を効率的に扱い、精度の高い結果を得ることができた。さらに、必要に応じて型情報を拡張することで、参照パターンや依存の解析に応用することができた。

また、多くの研究で使われている抽象解釈の代わりに、制約充足による解析を行った。KL1 は、並列実行や依存関係によるゴールの実行順の動的な変化、ストリーム通信による間接的かつ双方向性を持つデータフローといった特徴を持つため、大域的かつトップダウンの抽象解釈では、精度の高い結果を得ることが困難である。本手法では、個々のゴールが課す制約からボトムアップに解析を行うことにより、この問題を解決した。さらに、この手法はプログラム片に対しても解析が可能であるため、今後、分割コンパイルへの対応も容易に行えるものと期待できる。

この解析手法を利用して、以下に述べるノード間通信とゴール・スケジューリングの最適化を行い、本手法の有用性を実証した。

1.2.2 ノード間通信の最適化

KL1 の並列処理系のほとんどは、各計算ノードが固有のメモリ空間を持つ分散メモリ型の実装方式を用いている。また、ノード間でどのデータが参照されるかは実行時に決定されるため、ノード間データ参照は、実行系による要求駆動的なデータ転送により実現される。参照されるデータが構造型データである場合、従来の実装ではこの要求駆動方式を単純に適用し、要求ごとの細粒度通信を行う。その結果、データの大きさに比例した回数の通信が生じ、そのオーバヘッドが大幅な速度低下を引き起こす。

この解決方法として、構造型データの最外層が参照された時点で、全体を送信することにより通信回数を減らすという、バッチ転送モード [15] が提案され、実装されている。しかしこの方式では、構造型データに受信側で直接参照しない要素が混ざっている場合も、それらすべてを送信してしまう。このため、一つの構造型データを複数のノードが各々部分的に参照するようなプログラムでは、通信回数が減少しても実行を通しての総転送量は増加し、かえって速度低下を生じてしまう。

そこで本研究では、データ要求ノード側で必要な要素だけをまとめ送りするという、選択的一括送信手法を提案した。本手法は、まず型解析を利用し、各ノードでのデータ参照パターンを静的に求める。次に、この参照パターンに基づき、各々のノードで参照されるデータのみを送信バッファに格納する選択的一括送信コードを生成する。そして、実行時にデータ要求メッセージを受信すると、要求された値に対応する送信コードを実行することによって、必要な要素のみをまとめたメッセージを返信する。この手法によって、参照されない余分な要素を転送することなく、データのまとめ送りによる通信回数の削減が実現できる。

この選択的一括送信手法を用いて KL1 処理系 KLIC を改良し、性能評価を行った。この結果、従来の通信方式に比べ、通信回数が 2/3〜1/200 程度と大幅に削減された。また、総転送量についても、従来より減少するという効果が得られた。これは、選択的一括送信コードで送信要素を選択することにより、無駄な転送が生じるのを回避できたことに加え、通信回数を減らした結果、メッセージのヘッダ部の総量が減少したことによる。これらの効果により、実行時間についても、多くの場合は 2〜5 倍程度の高い性能向上が得られた。とくに、通信オーバヘッドが大きい環境

や、大規模なデータ転送が行われるプログラムでは、本手法の効果が大きく、従来の通信方式では得られなかった並列効果を引き出すことができた。

また、本手法はプログラムによっては効果がない場合があり、特殊な場合にはかえって性能低下を引き起こす例も見られた。そこで、この問題の解決法についても検討し、いくつかの具体策を提案した。

1.2.3 ゴール・スケジューリングの最適化

従来の実装では、ゴールを単位とする細粒度の並行制御を行うため、スケジューリングのオーバヘッドが非常に大きなものになる。本研究では、逐次化可能なゴール群をスレッド化することによって、並行制御の粒度を上げ、オーバヘッドを削減する手法を提案した。

本手法では、まず型解析を応用したデータ依存解析を行う。論理型言語では、構造型データの個々の要素を個別に具体化できるため、要素によって具体化・参照の向きが異なるなど、複雑なデータフローを生じる。また、変数間の同一化は両引数が未具体化であっても実行可能であるため、論理的なデータの流れが必ずしも実行上の先行制約とは一致していない。このため、ゴールの引数のデータフローを解析するような単純な方法では、正確な依存情報を得ることができない。そこで、本研究の解析手法では、型情報にデータを生成するゴールの情報を付加することによって、こうした問題を解決する。

そして、この依存情報を元に、実行順序を静的に固定可能なゴールについては、1スレッドとして逐次実行する。相互依存性のある部分はスレッドが分割されるため、これらのスレッドを動的スケジューリングすることにより、プログラムが持つ本質的な並行性は保たれる。また、本手法のスレッド化は、プログラムを変形するのではなく、スレッド内のゴールを逐次実行するように実行系を変更する。このため、ループのようにプログラム変形では逐次化不可能な部分も含め、大きなプログラム片を1スレッドにできる。さらに、各スレッドのゴール環境をスタック上に保持することで、ヒープの消費量を減らし、ガーベジコレクションのオーバヘッドを少なくすることができる。

また、並行性の粒度を上げることは、並列実行を行う場合、一部のノードのアイドル時間を増やし、かえって実行時間を増加させる可能性がある。そこで本研究では、他ノードより要求された値の生成スレッドを優先的にスケジューリングする、reply first 方式を提案した。この方式では、スレッド間で共有される変数に、解析情報を利用して自身を具体化するスレッドへのポインタを保持させる。そして、他ノードよりこの変数の値要求メッセージが届いたときには、このポインタを利用して、具体化スレッドをスケジューリングする。この結果、要求された値が最短時間で生成され、要求側ノードのアイドル時間を減らすことができる。

このゴール・スケジューリング最適化手法も、KLIC を改良する形で実装し、性能評価を行った。この結果、本手法によりゴールの中断を 1/2〜1/100 程度、ガーベジコレクションの回数を 1/3〜1/7 程度と、それぞれ大きく削減できることが確認できた。これに加え、スレッド化によりゴールキューの管理オーバヘッドもなくなるため実行時間も減少し、とくに従来方式ではスケジューリングオーバヘッドの大きかったプログラムでは、2〜3 倍程度の速度向上を得ることができた。また、並列実行でも同程度の速度向上が得られ、粒度向上によるアイドル時間増加の問題も、reply first 方式により解消されることが確認できた。

1.3 本論文の構成

本論文の構成は、以下に示すようになっている。

2章では、本研究の対象言語であるKL1、ならびにその処理系であるKLICについて説明する。まずKL1については、文法やデータ構造、組込述語といった基本機能について、概略を述べる。また、KLICについては、逐次／並列実行の実装方式や独自の拡張機能について述べる。さらに、KLICを始めとする従来のKL1の実装について、実行効率における問題点を論じる。

3章では、本研究で提案する最適化手法の基礎となるプログラムの静的解析について述べる。この章では、まず従来の様々な静的解析手法について、それらの目的や方法を比較する。続いて、本研究で提案する静的解析の基本的手法について詳しく述べる。ここでは、最初に基本となる型情報の定義や解析アルゴリズムを説明し、続いて実際のKL1プログラムを扱うための拡張について述べる。

4、5章では、この静的解析を応用した実行最適化手法を提案する。まず4章では、分散メモリ環境下でのノード間通信を最適化する手法を述べる。続いて5章では、逐次実行よりゴール・スケジューリングのオーバヘッドを削減する最適化手法を述べる。いずれの章においても、まず従来の実装における問題点を論じる。次に、これを解決する最適化手法を提案し、その概要を述べる。続いて、この最適化手法に必要な情報を得るための解析手法として、3章の手法を拡張する方法を説明する。そして、その解析結果を利用し、拡張・最適化されたコード生成を行う方法を述べる。また、それぞれの手法の実装とそれに基づく性能評価結果についても論じる。

最後に第6章で、本論文のまとめと今後の研究課題を述べる。

Chapter 2

並列論理型言語 KL1 とその実装

2.1 緒言

本章では、まず 2.2 節で、本研究の対象言語である KL1 について、文法やデータ構造などの概略を述べる。また、組込述語やマクロといった実用的な論理型言語としての拡張部分や、プログラミング技法についても、本研究に関連するものについて説明する。

この KL1 の処理系は過去にいくつかの実装がなされているが、主なものは Multi-PSI [16] や PIM [17, 18] といった、論理型言語専用に設計された並列計算機を対象としていた。最近では KLIC[14] と呼ばれる処理系が開発され、ワークステーションや様々な並列計算機上で利用できるようになっている。本研究で提案する手法も、この KLIC を対象として実装を行っている。そこで、KLIC についても 2.3 節で、逐次・並列の実行方式やデータ構造の実装方法などについて概観する。

KLIC をはじめとする過去の実装では、KL1 を効率的に実行するために様々な工夫がなされているが、手続き型言語などの実装に比べると、充分な性能を得ているとはいえない。本研究の目的は、静的解析情報を利用することにより、この問題を解決することにある。そこで、最後に 2.4 節で、現在の KLIC における実行効率上の問題点について説明する。

2.2 並列論理型言語 KL1

KL1 [11] は、Flat GHC (Flat Guarded Horn Clauses) [19, 20] に基づく、Commited Choice 型言語である。KL1 は第五世代コンピュータプロジェクトの核言語として、(財) 新世代コンピュータ技術開発機構 (以下 ICOT) で設計され、Multi-PSI [16] や PIM [17, 18] 上に実装されている。

2.2.1 文法

KL1 のプログラムは、以下の形をした節 (クローズ) の集合で表される。

$$H \ :- \ G_1,\ldots,G_m \mid B_1,\ldots,B_n. \quad (m \geq 0, n \geq 0)$$

H, G, Bを、それぞれクローズヘッド、ガードゴール、ボディゴールと呼ぶ。また、ガードゴールおよびボディゴールの並びを、それぞれガード部、ボディ部と呼び、両者の区切りを示す記号 '|' を、コミットメントバーと呼ぶ。

左右の記述は等価である。

```
p(X,Y) :- true | q(X,Y).    ⇔    p(X,Y) :- q(X,Y).
p(X,Y) :- true | true.      ⇔    p(X,Y).
```

図 2.1: 節の省略形

表 2.1: KL1 の標準データ型

データ型	例
整数	123
浮動小数点数	1.23
記号アトム	abc, []
リスト	[1,2,3], [a|b]
ファンクタ	f(a,b)
ベクタ	{1,2,3}
ストリング	"abc"

ゴールは述語の呼出であり、述語はヘッドの名前と引数の数が等しい節の集合で定義され、'述語名/引数の個数' の形で表記される。効率的な実行を可能にするため、ガードゴールには組込述語しか記述できないという制限が設けられている。ガード部またはボディ部に記述するゴールがない場合のために、常に成功する組込述語 true が用意されているほか、ガード部のみ、またはガード部およびボディ部を省略することもできる (図 2.1)。

また、ボディゴールは同時に実行することができ、これによってプログラムを並列実行するようになっている。並列実行ゴールの指定は後述のプラグマにより行う。

なお、以下の記述では説明のため、必要に応じて節に番号を振る。また、ゴールに番号を振る必要がある場合には、$p/1^i$, $p^j(X,Y)$, $X=^kY$ のような形で記述する。

2.2.2 データ構造

KL1 では、表 2.1に挙げるようなデータ型を提供している [21]。

整数や記号アトムなどのアトミックデータ型は内部構造を持たず、処理の基本単位となる値を持つ。これに対し、リストやファンクタなどの構造型データ型は、引数として他のデータ構造を持つことができる。このため、入れ子や再帰などの複雑なデータ構造を表現できるようになっている。

KL1 の変数は論理変数であり、初期状態では特定の値や型を持たない未具体化 (unbound) 状態である。実行中に変数と変数を同一化 (unification) することにより、両変数は論理的に同じ値を持つことになる。また、アトミックデータや構造型データと同一化 (具体化、instantiation) すると、そのデータを具体値として持つようになる。一度具体化された変数は、他の値に書き換えることはできない (単一代入規則)。以下、本論文ではこの論理変数を単に変数と呼ぶ。また、具体

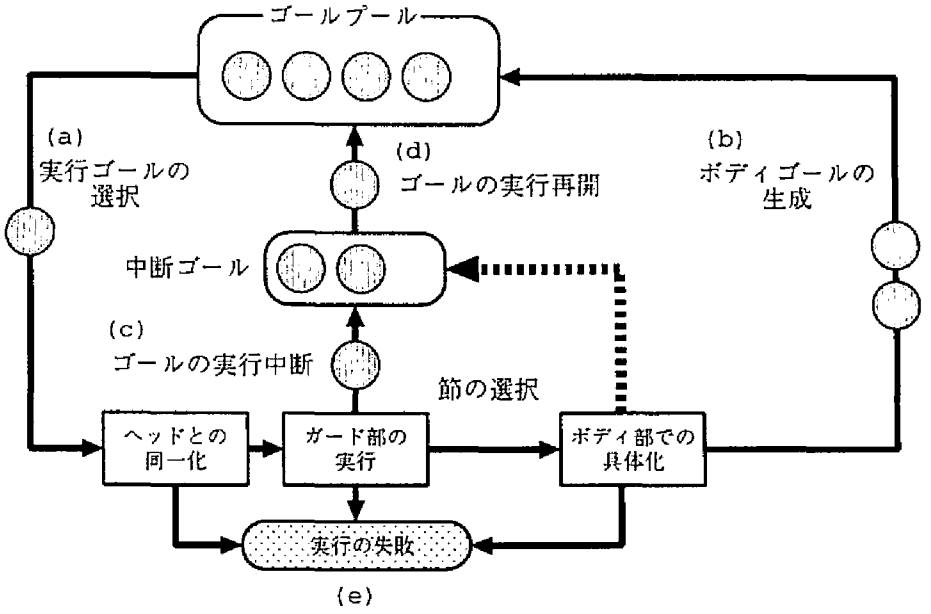


図 2.2: KL1 の実行モデル

値が構造型データである場合、その引数として未具体化変数を持つことができ、必要に応じて後からその部分を具体化できる。したがって具体値は、すべての要素が完全に具体化された基底項であるとは限らない。

2.2.3 実行モデル

KL1 の実行モデルは、図 2.2のようになっている。

KL1 での実行単位はゴールであり、実行可能なゴールはゴールプールに保持されている[1]。この中から実行ゴール Hを一つ選択し (図 2.2(a))、述語 Hを定義する各節のヘッド H_iとの同一化およびガード部 $G_{i1},\ldots,G_{im}$の実行を試みる。このとき、ガードゴールはヘッドの引数を具体化できない。続いて、成功した節 (複数ある場合は、その一つが非決定的に選ばれる) のボディ部 $B_{j1},\ldots,B_{jn}$を新たな実行可能ゴールとして生成し、ゴールプールに追加する (図 2.2(b))。この一連の操作を、ゴールのリダクションと呼び、リダクションを繰り返すことによってプログラムを実行し、ゴールプールに実行可能ゴールがなくなると終了する。このとき、まだ中断ゴールが残っていれば、それらの中断ゴールは永久に実行不能である。これを、実行のデッドロックという。

ガード部の実行に際して、ガードゴールが参照した変数が未具体化であった場合、実行中のゴールは実行を中断 (suspend) する (図 2.2(c))。ボディ部での組込述語によって変数が具体化されると、その変数への参照により中断していたゴールは実行を再開 (resume) できる。具体的にはゴールプールに戻され、再び実行ゴールとして選択されるのを待つ (図 2.2(d))[2]。

また、Prolog のようなバックトラックは行わず、ヘッドとの同一化やガード部の実行がすべて

[1] KLIC など多くの実装では、キューとして実現しており、本論文ではこれを実行可能ゴールキューと呼ぶ。

[2] 「ゴールの実行再開」とは、再び実行可能になるという意味であり、直ちにそのゴールが実行されることを意味しない。

失敗した場合、ならびにボディ部の組込述語の実行に失敗した場合は、プログラム自体が実行を失敗し、終了する (図 2.2(e))。

2.2.4 拡張機能

KL1 は OS などのシステム記述から、実用的なアプリケーション作成まで、幅広い利用を目的として設計されている。このため、Flat GHC と比べると、次のような機能が拡張されている [21]。

荘園

Flat GHC ではバックトラックが行われないため、ゴールの実行失敗はプログラム全体の異常終了を引き起こす。このような言語でシステム記述を可能にするため、KL1 には荘園という機能が設けられている。

荘園はプログラムのある部分 (ゴール群) をまとめたものである。特定のゴール群を荘園内で実行することにより、荘園内の実行や使用する計算機資源を制御したり、その例外発生をトラップしたりすることができる。KL1 で記述された OS である PIMOS [21] では、OS を構成するゴール群がユーザプログラムを構成するゴール群を荘園内で実行することにより、資源管理やシステムの保護を実現している。

Unix 上で動作する KLIC では、この機能は提供されていない。また、本研究はユーザプログラムを高速に実行できる処理系の実現が目的であるので、本論文では荘園機能は扱わない。

モジュール

KL1 プログラムはモジュールと呼ばれる単位に分割することができる。1 つのモジュールはモジュール名の宣言と述語定義節の並びから成り、他モジュールで定義された述語の呼出しは 'モジュール名:ゴール' の形で記述する。

この機能は KLIC でもサポートされており、1 モジュールが C の 1 関数に変換される。このため、個々のモジュールを別ファイルに分けることで、プログラムの分割コンパイルが可能になっている。

3.3 節で述べるように、本研究の最適化では、制約充足を用いたボトムアップの解析手法を用いている。したがって、個々のモジュールを解析し、得られた結果を融合することで、原理的には分割コンパイルに対応できる。

ただし、現在の実装では、部分的な解析結果の融合処理は実現されていない。このため、個々の述語をモジュール名と述語名の組によって区別することにより、プログラム全体の大域的な解析・コンパイルを行い、分割コンパイルには対応していない。

非対称な節の扱い

Flat GHC では節の記述順序は意味を持たないが、KL1 では記述の省略や実行効率の向上を目的として、次のようにリダクション時の節選択を制御できるようになっている。

1. `alternatively.`
   ある述語の定義節間にこの記述を挿入すると、それ以前の節が優先的に選択され、これらの節がその時点で選択できない (ガード部で失敗するか、必要なデータが揃っていない) 場合に、それ以降の節が実行されるようになる。

ただし、この優先順位は必ず守られるとは限らないため、実行効率向上のための指針としての役割しか持たない。

2. otherwise.
ある述語の定義節間にこの記述を挿入すると、それ以前の節すべてについてガード部で失敗した場合のみ、それ以降の節が選択される。未具体化変数の検査のように失敗するかどうか判明しない場合は、その場で otherwise 以降の節を選択することができず、リダクションしようとしているゴール自体が中断される。

プラグマ

ゴール間の優先度制御や並列実行の指定には、プラグマと呼ばれる表記を用いる。プラグマには次のようなものがある。

1. *goal*@priority(割合)
ゴールの優先度を指定する。他に相対的な優先度を指定する記法がある。

   このプラグマは通信最適化には無関係であるが、ゴールの実行順に影響するため、ゴール・スケジューリング最適化の際には考慮する必要がある。

2. *goal*@node(ノード番号)
そのゴールを指定したノードに送信し、そこで実行する。ノードとは共有メモリで結合された1個以上のプロセッサ集合のことである。

   このプラグマは物理的な並列実行の単位を示すものであるため、通信最適化において重要な意味を持つ。また、このプラグマつきのゴールは送信側でスケジューリングされないため、ゴール・スケジューリング最適化の際に別扱いにする必要がある。

組込述語

KL1 では、実用的なプログラム記述を可能にするため、多数の組込述語が用意されている。以下、主なものについて簡単に説明する。

1. atom(X), integer(X), list(X), functor(X,N,A) などの型検査述語
KL1 で提供されるデータ型について、それぞれ引数 X がその型であるなら成功する述語が用意されている。functor/3 は、引数 X が主ファンクタ名 N、引数個数 A であるなら成功する。

2. add(X,Y,Z), subtract(X,Y,Z) などの算術演算述語

3. equal(X,Y), less_than(X,Y) などの算術比較述語
整数と浮動小数点数には、各種の算術演算および比較を行う述語が用意されている。

4. new_functor(F,N,A), arg(P,F,A), setarg(P,F,E,NF) など
構造型データには、動的生成やその要素を操作する述語が用意されている。ファンクタの場合、new_functor(F,N,A) は、主ファンクタ名 N、引数個数 A のファンクタ (引数はすべて整数 0) を生成して F と同一化する。arg(P,F,A) は、ファンクタ F の第 P 引数を A と同一

化する。setarg(P,F,E,NF) は、ファンクタ F の第 P 引数を E に置き換えたものと、NF を同一化する。

ベクタや文字列についても、これと同様な述語が用意されている。

マクロ

プログラムの記述を容易にするため、定数マクロなどのユーザ定義マクロの他、条件分岐マクロや数式マクロなど、システム定義マクロが用意されている。

本論文では、基本的にマクロは前処理により展開済であるとする。ただし、記述を簡単にするため、以下のマクロ表記は必要に応じて使用する。

1. 算術演算マクロ
   整数、浮動小数点数について、それぞれ:=, :$=が算術演算を表す。右辺には+, -, *, / などを用いた数式を記述でき、その演算結果が左辺と同一化される。

   このマクロの展開結果は、右辺の数式を add/3 などの算術演算述語で展開し、最後の演算述語の結果をマクロの左辺と同一化する、というものになる。たとえば、マクロ X := Y+Z*2 は、ゴール群 multiply(Z,2,W0), add(Y,W0,X) のように展開される。

2. 算術比較マクロ
   整数、浮動小数点数について、<, >=などの不等式で算術比較を表す。両辺には算術演算マクロと同様に数式を記述できる。一致、および不一致は、それぞれ=:=,=\=で表す。

   このマクロの展開結果は、両辺の数式を add/3 などの算術演算述語で展開し、両者の演算結果を less_than/2, equal/2 などで比較検査する、というものになる。たとえば、マクロ X =:= Y+Z*2 は、ゴール群 multiply(Z,2,W0), add(Y,W0,W1), equal(X,W1) のように展開される。

2.2.5 プロセスとストリーム

KL1 における代表的なプログラミングスタイルとして、複数の並行プロセスを生成し、その間をストリームと呼ばれる通信路で接続する、という方法がある [15]。

論理型言語の実行単位となるゴールは、リダクションが行われると直ちに消滅する。このため、KL1 ではゴールの再帰呼出を用いて、一定期間にわたって存在し続けるようなプロセスを実現する。プロセスの内部状態はゴールの引数として保持され、新しい値を引数として再帰呼出を行うことによって、内部状態を変化させる。本論文では、ゴール p/n を開始ゴールとする並行プロセスを、並行プロセス p/n と呼ぶことにする。

並行プロセス間での情報交換は共有変数により行うが、論理変数を直接メッセージで具体化してしまうと値を変えられないため、2 回以上の通信を行うことができなくなる。そこで、プロセス間の通信路を、プロセス間のメッセージを要素とするリストとして実現する。これを、ストリームと呼ぶ。

出力側プロセスは、共有変数 (ストリーム変数) を、car が渡したい具体値 (メッセージ)、cdr が未具体化変数であるようなコンスセルに具体化する。以後、cdr 部を新たな共有変数とし、出力側プロセスが同様にこれをコンスセルに具体化することによって、通信を繰り返すことができる。ストリームが不要になった場合は、通常、ストリーム変数を [] で具体化する。これを「ストリームを閉じる」と表現する。

```
main :- p(10000,S), c(S).
p(T,S) :- T>0 | S=[T|S0], T0:=T-1, p(T0,S0).
p(T,S) :- T=0 | S=[].
c(S) :- S=[T|S0] | c(S0).
c(S) :- S=[] | true.
```

図 2.3: プロセスとストリーム

```
main :- p(10000,S), c(S,dummy).
p(T,S) :- T>0 | S=[put(T)|S0], S0=[get(T0)|S1], p(T0,S1).
p(T,S) :- T=0 | S=[].
c(S,T) :- S=[put(T0)|S0] | c(S0,T0).
c(S,T) :- S=[get(T0)|S0] | T0 := T-1, c(S0,T0).
c(S,T) :- S=[] | true.
```

図 2.4: ストリームによる双方向通信

入力側プロセスは、ガード部でストリーム変数を参照することにより、(1) 変数が未具体化なら具体化されるまで待ち、(2) コンスセルに具体化されれば car のメッセージを処理し、(3) [] に具体化されれば入力が終了したとみなす、という動作を実現できる。また、(1) の動作は KL1 における未具体化変数参照時のゴール中断機構により行われるため、プロセス間の同期が自動的に実現される。

図 2.3に、このプログラミングスタイルを用いた簡単な例を示す。このプログラムでは、並行プロセス p/2 と c/1 が生成され、両者の間がストリームにより接続される。p/2 は内部状態を表す変数 T の値を 1 ずつ減らしながら、ストリームに出力していき、T が 0 になるとストリームを閉じて終了する。c/1 は入力ストリームの内容を次々に読み込み、ストリームが閉じられると終了する。

また、このストリームを用いた通信では、一本のストリームで双方向通信を実現することができる。たとえば図 2.4のプログラムでは、図 2.3と同様、並行プロセス p/2 と c/2 の間がストリームで接続され、p/2 から c/2 にメッセージが次々に送られる。図 2.3の場合と異なるのは、メッセージが put/1, get/1 のような構造型データとなっており、引数として未具体化変数を持てるようになっている点である。put/1 の場合は、引数も p/2 により整数に具体化され、c/2 に送られる。しかし、get/1 の場合は、引数は未具体化変数のまま c/2 に送られ、c/2 がこれを整数に具体化する。この結果、p/2 がその値を参照できる。このように、メッセージに未具体化変数を含ませることで、ストリームとは逆方向のデータ送信を行うことができる。

2.3 KL1 処理系 KLIC

KLIC [14, 22, 23] は、ICOT で開発された KL1 処理系である。KLIC では KL1 プログラムをいったん C プログラムに変換し、ターゲットマシンのオブジェクトコード生成は、その計算機用の C コンパイラに行わせる方式を取っている。このため、物理通信などの機種依存部を除き、移

表 2.2: KLIC のタグ付セル

タグ		表すデータ
REF	(00)	変数
CONS	(01)	コンスセル
FUNC	(10)	ファンクタ
ATOMIC	(11)	記号アトム／整数

植性の高い処理系になっている。現在、汎用並列通信ライブラリである PVM [24] を利用した並列版が、ワークステーションや様々な並列計算機環境上で利用可能になっている。

実行プログラムは、上記の方式で得られたオブジェクトと、ランタイムライブラリとして用意された実行系をリンクすることで得られる。ゴールのスケジューリングやガーベジコレクション (GC) は、この実行系中の処理系核と呼ばれる部分が行う。

2.3.1 データ構造の実装

KLIC では 1 ワードを 1 セルとし、このうち 2 ビットをタグ領域として、そのセルの内容を表している (表 2.2)。

KL1 のデータ型のうち、記号アトムと整数はまとめて ATOMIC タグの付いたアトミック型として扱われ、さらにもう 1 ビットのタグを設けて区別する。ファンクタは FUNC タグ付セルで表されるが、ファンクタの一種であるコンスセルは CONS タグ付セルで区別され、リストを表現するのに使われる。浮動小数点数、ベクタ、ストリングは、2.3.3 項で説明するジェネリック・オブジェクトとして実装されている。

また、変数は REF タグつきのセルで表され、内部に値のアドレスが格納される。このようなセルを参照ポインタと呼ぶ。未具体化変数は、自分自身のアドレスを格納することで、値を持たないことを表す。構造型データは、引数にこの変数を持つことができ、他のデータ構造を指すことで入れ子になったデータを表現したり、未具体化変数を持つことで要素の一部が未具体化のデータ構造を生成したりすることができる。

これらのデータ構造はすべてヒープ上で生成・保持され、ヒープはコピー型ガーベジコレクションにより管理されている。コピー型ガーベジコレクションでは、事前に同サイズのメモリ領域が二つ確保されており、データ生成はその一方だけを用いる。使用中の領域が一杯になると、処理系核のガーベジコレクションを行うルーチンが起動し、生成済のゴールから指されているものなど必要なデータだけを、もう一方の領域にコピーする。それ以後はコピー先の領域をデータ生成に使用し、再び領域が一杯になれば、今度は逆向きに同様の処理を行う。

2.3.2 逐次版の実行方式

逐次版 KLIC の実行は、図 2.5のようになっている。

実行の単位であるゴールは、ゴールレコードとして表現され、次に実行されるゴールのゴールレコードへのポインタ (以下 GP)、自身に対応する実行コードへのポインタ (以下 CP)、および引数の並びからなる。このゴールレコードもデータ構造と同様、KLIC のヒープ上に生成され、不要になったゴールレコードの領域は、ガーベジコレクションの際に解放される。

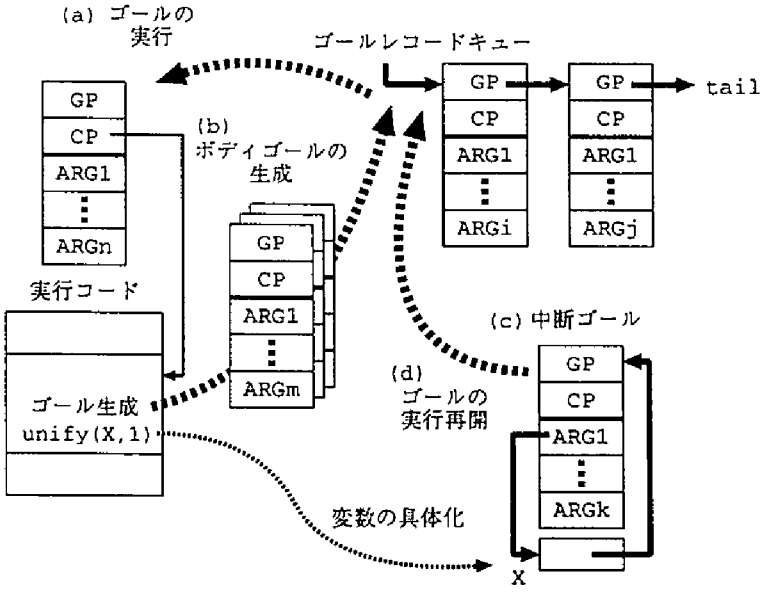


図 2.5: 逐次版 KLIC の実行

ゴールプールはゴールレコードキューとして実現されており[3]、ゴールのリダクションは次のように行われる。まず、ゴールレコードキューの先頭レコードが取り出され、CP が指すコードが実行される (図 2.5(a))。実行コードは、ユーザの記述した KL1 プログラム (ユーザ定義述語の定義節の並び) にしたがって、ヘッドとのパターンマッチングやガード部の検査を実行し、成功した節のボディゴールに対応するゴールレコードを生成してゴールレコードキューの先頭に挿入する (図 2.5(b))。

ガード部の検査で、参照した変数が未具体化の場合、実行中のゴールは中断する。このとき、中断原因となった未具体化変数の参照ポインタを書き換え、中断したゴールのゴールレコードを指すようにする。すでに他のゴールレコードを指している場合は、GP を使って、ゴールレコードのキューを形成する。これを、「ゴールが変数にフックする」という (図 2.5(c))。これによって、中断ゴールのゴールレコードはゴールレコードキューから外された状態になるため、中断原因が解決する前に再び実行されてしまうことはない。

のちにこの変数が具体化されたとき、処理系核はその参照ポインタがゴールレコードを指しているかどうか調べ、フックしている中断ゴールレコードが存在する場合は、それらをゴールレコードキューの先頭に挿入する (図 2.5(d))。これによって、中断ゴールの実行再開が実現される。

なお、KLIC では、実行開始時の初期ゴールとして `main/0` が与えられる。

2.3.3 ジェネリック・オブジェクト

データ型とその操作を追加する機構として、ジェネリック・オブジェクトが提供されている。ジェネリック・オブジェクトは、表すデータの値など内部情報を格納するデータ領域と、メソッドと呼ばれるデータ操作用の C コードから成る。KLIC の処理系核は、対象がジェネリック・オ

[3]図では省略しているが、`@priority` により優先度が指定されている場合は、同じ優先度のゴールレコード毎に、別個のキューを形成する

ブジェクトならば所定のメソッドを呼び出すことで操作を行い、メソッドやデータの内容には関知しない。これによって、カーネル部分を変更することなく機能を拡張できるようになっている。

ジェネリック・オブジェクトには、データ・オブジェクト (data object)、コンシューマ・オブジェクト (consumer object)、ジェネレータ・オブジェクト (generator object) の 3 種類がある。

データ・オブジェクトには特別な機構は設けられておらず、データ型を拡張する手段として用いられる。KLIC では、標準データ型のうち浮動小数点数、ベクタ、文字列をデータ・オブジェクトとして実装している。

コンシューマ・オブジェクトとジェネレータ・オブジェクトは変数にフックし、これらの変数に具体化や参照などが生じたとき、所定のメソッドが自動的に呼ばれるようになっている。一般には、コンシューマ・オブジェクトに対して具体化が行われると UNIFY メソッドが呼ばれ、その具体値に対し何らかの処理を行う。一方、ジェネレータ・オブジェクトに対して参照が行われると、GENERATE メソッドが呼ばれ、何らかの値を生成して参照した側に渡す。KLIC では、複数のストリームをマージするマージャや乱数発生器などに、これらのオブジェクトを用いている。また、次項で述べる分散メモリ並列版でのノード間通信機構の実現にも、ジェネリック・オブジェクトが用いられている。

2.3.4 分散メモリ並列版の実行方式

分散メモリ並列版 KLIC の計算モデルは、共有メモリを持たない複数のノード上でそれぞれ逐次版の実行系が動作し、互いにメッセージ通信をしながら処理を行うというものである [23]。

実行開始時点で、逐次版と同様に、あるノードに対して初期ゴール `main/0` が与えられる。並列実行は `@node` プラグマを用いて明示的に記述する。$goal(X_1,\ldots,X_n)$`@node`(N_c) という形のボディゴールがノード N_p 上に現れると、このゴールはノード N_c に送られ、そこで実行される。このとき、未具体化変数 X_i については、N_p 上の変数を指す外部参照ポインタ[4]が N_c に送られる (図 2.6(a))。外部参照ポインタはジェネレータ・オブジェクト EXREF として実装されており (図 2.6(b))、ノード間の参照や具体化をこのオブジェクトのメソッドに処理させることにより、処理系核の移植性を高めている。

N_c 上で X_i の値が参照されると、EXREF の GENERATE メソッドが呼び出される。このメソッドは、N_p に EXREF が指す値を要求する read メッセージを送り、EXREF を返信メッセージ処理用のコンシューマ・オブジェクト READ_HOOK に変化させる。また、X_i を参照したゴールは、逐次版と同様に変数にフックして中断する (図 2.6(c))。

N_p では、read メッセージを受信すると、answer メッセージにより要求された値を送信する。もし X_i が未具体化であれば、返信用のコンシューマ・オブジェクト REPLY_HOOK を X_i にフックしておく (図 2.6(d))。後者の場合、X_i が具体化された時点で REPLY_HOOK の UNIFY メソッドが呼び出され、値の送信処理を行ってから REPLY_HOOK を消滅させる。(図 2.6(e))。

N_c が answer メッセージを受信すると、READ_HOOK の UNIFY メソッドが呼び出される。このメソッドは、中断ゴールを実行再開させ、ノード間ガーベジコレクションのために外部参照ポインタの解放を知らせる release メッセージを N_p に送信し、READ_HOOK を消滅させる (図 2.6(f))。read メッセージを送信した時点で外部参照ポインタを解放できないのは、answer メッセージが到着する前に N_c 上の他のゴールが X_i を具体化する可能性があるためである。

[4]実際には、参照ポインタの値は実アドレスではなく、参照先のノードが持つ輸出表のエントリを指している。これは、ガーベジコレクションによる参照先の移動に対応するためであるが、本論文の議論には影響しないので、ここでは省略している。

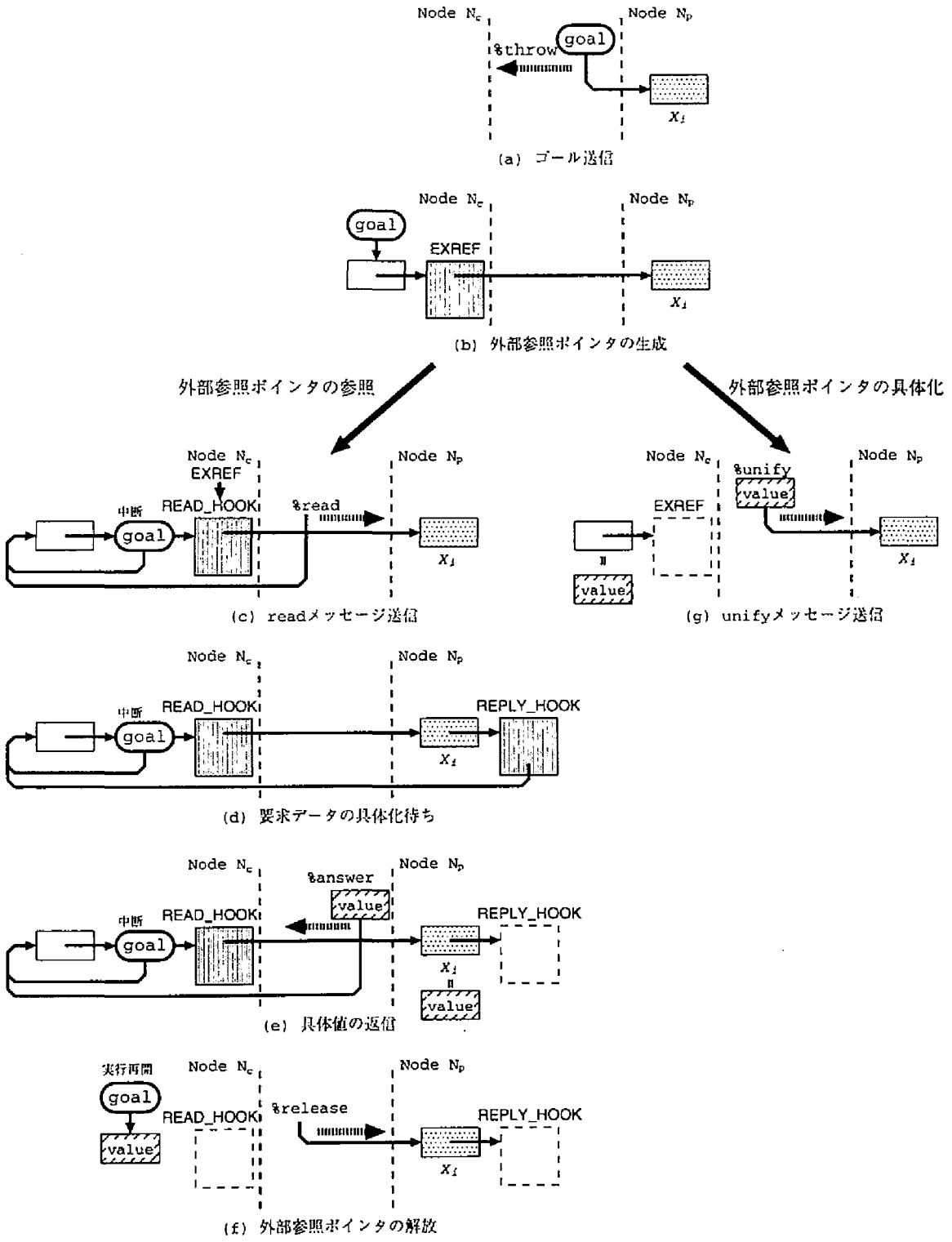


図 2.6: 分散メモリ並列版 KLIC のノード間通信

また、N_c上でX_iを具体化した場合は、EXREF の UNIFY メソッドが呼び出される。このメソッドは、N_pにこの具体値との同一化を要求する unify メッセージを送信し、EXREF を消滅させる。(図 2.6(g))。

以上のようにして並列実行を行い、すべてのノード上で実行可能なゴールがなくなると、実行系がこれを検出し、プログラムの実行が終了する。

2.4 従来の実装における問題点

KLIC では実行コード生成を C コンパイラが行うので、低レベル部分の最適化に関しては効率の良いコードが得られる。このため、逐次版 KLIC では、既存の論理型言語処理系に比べ、比較的高い性能が得られている [14]。

しかし、KL1 の実行モデルによる速度低下は解決できておらず、次に述べるような問題点を生じている。

2.4.1 ゴール・スケジューリングのオーバヘッド

2.3.2 項で述べたように、KLIC では未具体化変数への参照がおきると実行ゴールが中断し、ゴール・スケジューラは他の実行可能ゴールに制御を移す。この未具体化変数が他のゴールにより具体化されると、中断ゴールは実行を再開する。こうした中断・再開機構により、ゴールが正しい順序で実行されることが保証されるが、この処理が頻繁に行われると、そのオーバヘッドが無視できなくなる。また、ゴール環境 (KLIC ではゴールレコード) をヒープ上に確保する処理や、生成されたゴールを実行可能ゴールキューにつないだり、次の実行対象としてキューから取り出したりする処理も、オーバヘッドとなる。

2.4.2 ノード間通信のオーバヘッド

2.3.4 項で述べたように、分散メモリ並列版 KLIC では、ジェネリック・オブジェクトによりノード間通信を実現している。この機構により、明示的に通信を記述しなくとも、必要に応じて実行系がノード間でデータの転送を行い、逐次と同じプログラムが並列環境でも正しく動作することが保証されている。

しかし、現在の通信機構では、ノード間の細粒度通信が頻発し、そのオーバヘッドが大幅な速度低下を引き起こす場合が少なくない。これは、通信の相手や対象データが動的に決定するために、個々のデータ要素が必要になった時点で送信を要求するという、要求駆動的なデータのやりとりを行うことが原因である。この結果、リストやファンクタなどの構造型データを扱う場合、どの部分を参照するかが事前に判らないため、個々の要素が参照される度に通信を行ってしまい、通信が細粒度化する。

論理型言語では実行時に決定する要素が多く、実行時に動的な処理を行うため、上記のようなオーバヘッドを生じる。したがって、これを軽減するには、静的解析によってコンパイル時にプログラムの振る舞いを予測し、動的処理の一部を静的な手法に置き換える、という方法が有効である。

2.5 結言

以上に述べたように、KL1 は抽象度の高い記述により、複雑なデータ構造を容易に扱うことができ、記号処理・知識処理分野のプログラム記述に適している。また、実行モデルが論理的に並列であり、通信・同期が暗黙に行われるため、並列処理にともなうバグを混入させにくいという利点を持つ。さらに、Flat GHC をベースに様々な機能拡張を行ったことにより、実用的なプログラミング言語としての使用に耐えるものになっている。

最近では、KL1 処理系 KLIC が、ワークステーションなど多くの逐次・並列環境で利用可能になっている。KLIC は C へのトランスレータ方式を取ることで高い移植性を実現しており、また、コード生成にターゲットマシンの C コンパイラを使用するため、C コンパイラによる低レベル部分の最適化機能を利用できる。さらに、ゴールの実行制御やガーベジコレクションといった KL1 の基本機能をライブラリの形で提供し、ユーザのプログラムとリンクすることで、同期やメモリ管理などの低レベル部分が自動的に処理されユーザのプログラム記述が容易であるという、KL1 の利点を保っている。

しかしながら、現在までの KL1 処理系は、C のような手続き型言語の処理系に比べ、実行効率が劣る。KLIC でもこの問題は解決しておらず、とくに逐次実行部における動的ゴール・スケジューリングのオーバヘッド、ならびに分散メモリ並列版での細粒度通信オーバヘッドは、大きな速度低下の要因になっている。そこで、静的解析情報を利用してこれらのオーバヘッドを削減することが、本研究の目的である。

以下、3 章では、論理型言語のプログラム解析手法について論じ、本研究で用いた KL1 プログラムの静的解析手法を説明する。続く 4,5章では、この解析手法を基に、上記の問題点を解決する最適化手法について述べる。

Chapter 3

静的解析手法

3.1 緒言

本章では、KL1 プログラムの静的解析手法について述べる。

前章で述べたように、並列論理型言語 KL1 はその言語上の特徴により、実用的な並列プログラム記述に適した言語となっている。また、その処理系の一つである KLIC は、C へのトランスレータ方式の採用により、数々の並列計算機上で動作する移植性の高さと、C コンパイラによる低レベルの最適化が大きな長所となっている。

しかし、現在の KL1 の実装は、様々なオーバヘッドにより、多くの手続き型言語の実装に比較して、実行効率が劣るものになっている。この主な原因として、ゴールの実行順序やノード間の通信タイミングなど、実行時に処理される部分が多いという点が挙げられる。そこで、本研究では静的解析情報を利用して、従来の実装で動的に処理されていた部分を、ある程度までコンパイル時に解決してしまうことで、この問題を軽減することを目的としている。

以下、まず 3.2 節で、従来の研究で提案されてきた論理型言語の静的解析手法について概観し、比較検討する。続いて、3.3 節で、本研究で用いた解析手法を、詳しく説明する。

3.2 論理型言語の静的解析手法

2.2 節で述べたように KL1 は、同期・通信が暗黙に行われる、柔軟なデータ構造の記述が可能である、といった、プログラムを記述するユーザの負担を減らす上で有用な性質を持っている。しかしその反面、ゴールの実行順や変数の型など、プログラム上で明示されていない部分が多いために、単純な実装ではこうした不確定部分を動的に処理することになり、効率の低下を招く。KLIC など従来の実装では、2.4 節で述べたように、ゴール・スケジューリングやノード間通信のように実行時まで決定しない要因が多数存在し、その動的処理のオーバヘッドが速度低下の大きな原因となっている。

実際には、これらの動的処理は、必ずしも実行時に行う必要はない。たとえばゴールの実行は、依存によるデッドロックを生じないように、すべてのゴールについて実行順が動的に決定されるようになっている。しかし、あるゴールの集合について、デッドロックを生じない順番が判明していれば、コンパイル時に固定してしまっても問題を生じない。

この種の問題を解決するには、コンパイル時にプログラムに対する静的解析を行って実行時の振る舞いを調べる、という手法が有効である。この結果、決まった動作しか行わないと判明した

部分については、それに応じて実行コードを最適化し、不必要な動作やチェックをなくすことができる。ただし、プログラムが持つ非決定性や解析コストなどの問題より、完全な解析を行うことは困難であって、結果は一般に近似となる。したがって、まず、解析結果が正確 (accurate) ではなくとも正しい (correct) ことを保証する必要がある。その上で、目的に応じて、必要な情報に関する精度をなるべく高くする手法が求められる。

この種の静的解析の利用はKL1に限らず、他の論理型言語や関数型言語のように論理的な実行モデルに動的処理部分が多い言語一般に有効であり、従来より多くの研究がなされている。本節では、論理型言語を対象としたものを中心に、主な手法と応用分野について述べる。

3.2.1 静的解析の目的

プログラムの静的解析を行う目的としては、以下のようなものが挙げられる。

- コンパイル時の最適化
  本研究のように、静的解析によりプログラムの挙動を事前に調べ、それを利用して実行コードを最適化する。具体的には、変数が取り得る型を調べることで同一化処理での型判定を簡略化する手法 [25] や、変数の再利用性を調べることで不要になった領域をそのまま他のデータ領域に流用し、ガーベジコレクションの頻度を下げる手法 [26, 27] などがある。また、依存解析を利用したゴール実行の最適化 [28, 29] も提案されている。

- プログラムの検証やデバッグ
  データフローや変数の型に矛盾がないか調べることで、プログラムに誤りがないか検証したり、プログラムの停止性を証明したりすることができる。また、プログラムが正しいことを示すだけではなく、作成中の、おそらく誤りを含むプログラムに適用することによって、誤りの場所や原因を特定でき、デバッグの支援にも利用できる [30, 31, 32]。

- 並列性の抽出
  ゴール間の変数共有情報を抽出し、互いに依存のないゴールを並列に実行することで、AND並列性を引き出すことができる [26, 33, 34]。

3.2.2 静的解析の内容

従来の研究における静的解析の内容には、以下のようなものがある。

型解析

論理型言語などでは、変数が型を持たないため、通常は型宣言のような構文も存在しない。型解析では、そうした言語のプログラムに対し、プログラムの実行において変数が取り得る型を求めることを目的とする。

Milnerの研究 [30] では、型検査アルゴリズムを提案し、メタ言語MLを対象とした実装を行っている。この手法では、再帰を含む型の定義はユーザが明示的に与えなければならないという欠点があり、この種の型が頻発する論理型言語への適用には問題があった。

これに対し、再帰を含む型を扱える手法が、いくつか提案されている。Mycroftらの研究 [31] では、節表現に類似した型宣言の形式を導入し、Prologを対象に、バグ検出のための型検査を行っている。Mishraらの研究 [35] やZobelの研究 [32] では、任意の型をプログラムから自動的に抽出できる手法を提案している。前者では変数の型を、木構造の項の集合を表せるように正規表現

を発展させた、正規木を用いて表現している。後者ではPrologプログラム中の項をそのまま型として用い、型規則を導入して型表現を定義している。またいずれの手法も、型変数を導入して再帰的な構造を表している。

Bansalらの研究 [36] では、型変数を用いた型表現により再帰構造を扱うことができるのに加え、型で表現された抽象項に対する抽象同一化 (abstract unification) を定義して、後述する抽象解釈を行っている。これによって、同一化によって異なる変数が同じ値を持つ別名化 (aliasing) の問題を、自然に扱っている。また、次に述べるモード情報を合わせて求めることによって、プログラム中の生産者・消費者関係の解析を行っている。

モード解析

多くの論理型言語では、データフローの方向も明示的に定義されず、ゴールの引数は入出力の区別を持たない。モード解析では、このデータフロー情報を得ることを目的とする。ただし、単なる引数のモード情報だけでは、構造型データを介したデータフローが抽出できない。このため、型情報と組み合わせる手法も用いられている。

Somogyiの研究 [37] では、型にモードを付加する形で解析を行うことで、構造型データの引数についてもモードを得られるようにしている。またモード情報は、単なる入出力だけではなく、具体化されてから入出力される場合と、データフローの方向は入力または出力であるが未具体化のまま同一化される場合を、それぞれ区別して扱っている。各手続きのモード宣言はユーザが記述し、解析系は全体に適合するモード割り当てを行うようになっている。

Bruynoogheらの研究 [26] では、モード情報により同一化の振る舞いを調べ、不要になった領域の再利用によるガーベジコレクションの回数削減を行っている。また、変数間の共有情報を利用してAND並列性の検出が行えることも示している。

Debrayらの研究 [25] では、モードは入出力方向ではなく、実行の各段階での変数の具体化状況を表している。この手法では、変数の別名化を追跡することで、解析の精度を高めている。また、モード情報の利用方法として、不要な検査を省略して特化した同一化コードの生成や、節の融合による非決定性の削減などを提案している。

上田らの研究 [38, 39] では、プログラム全体のモード情報を、モードグラフの形で表現している。構造型データもこのグラフの構造の一部として含まれるため、その引数に関するモードも表現できる。また、この手法のモードは、基本的に時刻に依存しないという特徴がある。

依存解析

KL1などの並行／並列論理型言語 [40] では、並行実行されるゴールの実行順序や物理的に並列実行されるゴールの決定などが動的に行われるものが多く、速度低下の原因となっている。ゴール間の依存情報を利用することで、最適な実行順序や並列実行するゴールを決定したり、冗長な同期のオーバヘッドを削減したりすることができる。

Codishらの研究 [28] では、データフロー解析からゴール間の依存関係を求め、中断する可能性のあるゴールを識別する手法を提案している。

Kingらの研究 [29] では、同じ節内のゴールの間の依存関係を調べ、スレッドとして順序付けを行うことで、ゴール・スケジューリングのオーバヘッドを削減している。5章で述べる、本研究のゴール・スケジューリング最適化手法も、これに似た方法を用いている。

Muthukumarらの研究 [34] では、AND並列性を抽出するための依存解析を行っている。この手法では、変数間の大域的な共有関係を解析することによって、互いに依存しないゴールについ

ては実行時の依存や引数の具体化検査を省き、オーバヘッドを削減している。

粒度解析

これまでに述べた解析は、基本的に定性的な情報を得るものであるが、動的負荷分散処理では、プロセスの処理時間のように定量的な情報を必要とする。そこで、こうした部分の最適化のために、プロセスの粒度を静的に解析する手法が提案されている。たとえばKL1では、Tickの研究 [41] などがある。こうした手法では、入力データに対する処理時間のオーダを解析し、得られた近似式に実行時の値を当てはめることによって、少ないオーバヘッドで負荷量の予測を行う。

3.2.3 静的解析の手法

こうした解析情報を得る手法としては、次のようなものが提案されている。

抽象解釈

解析手法としては、Cousotらの研究 [42] を始めとして、抽象解釈 (abstract interpretation) を用いているものが多い [25, 26, 27, 28, 33, 34, 36]。

抽象解釈は、実際のプログラム上のデータ空間とそれに対する操作を、それぞれ抽象化されたデータとそれに対する操作に置き換え、プログラムの実行をシミュレートする。たとえば文献 [42] で用いられている例では、プログラムにおける整数空間を抽象空間 $\{(+),(-),(\pm)\}$ にマッピングし、この空間に対する演算を定義する。これによって、実際の計算 $-1515*17$ を計算する代りに抽象演算 $(-)*(+) \Rightarrow (-)$ によって、計算結果が負の整数値であることが判る。しかし、抽象化によってある程度の情報が失われるため、誤りではないが正確でもない結果が得られる場合がある。たとえば上記の例で加算を抽象解釈すると $(-)+(+) \Rightarrow (\pm)$ となり、あまり意味のある情報は得られない。この抽象解釈の結果の正確さは、抽象空間の取り方で変わってくるが、実行時間とのトレードオフになっている。そのため、各研究において様々な抽象空間の定義が提案されている。

論理型言語の抽象解釈では、ゴールの呼出木をトップダウンにたどる方法がよく用いられている。この場合、抽象実行の各過程について、変数の (抽象領域における) 束縛状態などを表すパターンが定義される。ゴールの抽象実行は、ゴール呼出前の呼出パターン (call pattern) に対し、呼出後の成功パターン (success pattern) を計算することを意味する。あるゴールの抽象実行後、その成功パターンを次ゴールの呼出パターンとして実行を続けることで、実行の各段階における状態を得ることができる。

また、プログラム中に再帰呼出がある場合、単純に抽象実行を行うと無限ループに陥ってしまう。これに対し、Muthukumarらの手法 [34] では、まず再帰呼出なしの実行を行い、成功パターンをメモテーブル (memo table) に記録しておく。次にこの値を呼出パターンとして再帰呼出を行い、得られた成功パターンでメモテーブルを更新する。以後、これを繰り返すごとに成功パターンがより正確になり、やがて不動点 (fixpoint) に到達する。

制約充足

一方、上田らの研究 [38, 39] では、個々の操作により与えられるモード制約を考え、それらすべてを満たすモードグラフを生成するという手法で、モード解析を行っている。この手法は、抽象解釈のような大域的な解析を行うのに比較して、不完全なプログラムを扱うことができるため、

分割コンパイルに対応しやすいという利点がある。また、抽象解釈のように、再帰構造に対して不動点の計算を繰り返す必要がない。

3.3 KL1 の型解析手法

本節では、前節で概観した論理型言語の静的解析手法を踏まえ、本研究で用いた KL1 の解析手法を説明する。

3.3.1 手法の特徴

本研究では、プログラムの挙動の解析手段として、型解析を採用した。これは、次の理由による。

1. ノード間通信の最適化に必要なノードごとの参照パターンを調べるには、型情報が必要である。一方、ゴール・スケジューリングの最適化に必要なゴール間の依存情報については、モード情報が使われることが多い。しかし、どのような種類の値を参照する場合にどのゴールに依存するかといった、より詳細な情報を得るには、型情報も合わせて必要となる。

   上田らのモード解析手法 [38, 39] は、本研究の手法とは逆に、構造型データ中のモードを表すために、モード情報中に一種の型情報も含ませている。この表現方法では、型情報を取り出すためにモードグラフをたどる必要があるため、本研究の目的には不便である。

2. モードや依存ゴールなどの情報を、型と関連づけて表すことにより、型解析を拡張する形で他の情報も解析できる。このため、本研究のように複数の最適化手法に用いる場合に都合が良い。

また、3.2.3 項で述べたように、従来より提案されてきた型解析手法の多くは、抽象解釈を用いている。これに対し、本研究では上田らのモード解析手法と同様に、組込述語やボディ・ヘッド間の同一化により課せられる制約より、各変数が取り得る型の集合を求める、という方法を用いる。これは、以下の理由による。

1. 左深さ優先でゴールの呼出木をたどる Prolog と異なり、KL1 では変数参照に起因するゴールの中断・再開によって、ゴールの実行順が動的に変化する。このため、実際の値を求めずに実行をシミュレートする抽象解釈で、精密な結果を求めることは難しい。一方、KL1 にはバックトラック機構がなく、ガードによって変数の取り得る値に制限が加わるため、制約問題として解きやすいと言える。さらに、並列実行による実行時の環境変化や、多用されるストリーム通信による間接的なデータフローの発生、といった KL1 の特徴も、ボトムアップに結果を求める制約伝搬を用いた方が扱いやすい。

2. トップダウンでの抽象解釈を行う場合、あるゴールの実行開始前の状態に対し実行後の状態を求める、という計算を繰り返す。このため、抽象解釈開始時点での初期状態を与える必要がある。一方、本研究で用いた制約伝搬による解析手法は、実行開始時点の状態を必要としない。このため、4.4 節で述べる並列プロセス単位の解析のように、プログラムを分割して解析できる。

3. 本研究の最適化手法で必要な情報は、実行の過程全体を通した大域的な型やデータフローの情報であり、実行時の文脈に依存した情報は必要としない。このため、抽象解釈のように実

表 3.1: 型解析におけるデータ型

アトミックデータ型		構造型データ型
記号アトム	a	$s(name, arity, (T_1, \ldots, T_n))$
整数	i	
浮動小数点数	f	
文字列	s	

際のプログラム実行の流れに沿う解析を行う必要はなく、課せられる制約から大域的な情報を求めれば充分である。

以下、本研究の解析手法を詳しく説明する。

3.3.2 型情報

2.2.2 項で述べたようにKL1の変数は特定の型を持たない。しかし、一度具体化されると値を変更できないから、この具体値の型を、その変数の型と考えることができる。

ただし、同じ変数でも実行パスにより異なるデータ型の値に具体化されることがあるから、静的解析では一般に変数の型は一意に決まらない。そこで、本手法では変数の型を、その変数が実行時に取り得る具体値の型の集合と定義する。本論文ではこれを変数の型集合と呼び、次のように定義する。

型集合の定義

本解析手法では変数 v_i の型集合を、$T(\mathrm{v}_i) = \{e_1, \ldots, e_n\}$ と定義する。各 e_j は型 t_j、型変数 v_j、不明要素*のいずれかである。

t_j 表 3.1のいずれかの値をとる。

浮動小数点数や文字列は実際にはアトミックデータではなく、KLIC ではジェネリック・オブジェクトとして実装されている。しかし、これらのデータ型は、専用の組込述語以外ではその一部を取り出したり変更したりすることができない。そこで、本解析手法ではアトミックデータの一種として扱う。

構造型データ型については、名前や引数の個数が異なるファンクタを、すべて異なる構造型とみなす。これは、同じ変数でも、どのようなファンクタに具体化されるかによって、参照される要素の位置や深さが異なることがあり、解析情報でもこうしたファンクタを区別する必要があるからである。コンス・セルやベクタについても、特殊な名前を持つファンクタとみなす。以下、名前 f、引数個数 n である構造型データ型を、$t_{f/n}$ と表記する。

また、構造型データ型の各引数については、実行パスにより異なる型に具体化される可能性がある。このため、それぞれの引数を、再帰的に型集合 T_i で表す。

v_j プログラムの記述中に現れる構造型データの引数に変数が含まれる場合、構造型データ型の対応する引数を、型変数で表す。型変数はプログラム変数と一対一対応しているので、本論文で

はプログラム変数 v_i に対し型変数 v_i のように、前者をタイプライター体、後者をイタリック体で表記する。

プログラム変数 v_i が実際に取る型は、対応する型変数 v_i の型集合によって与えられるが、本手法では型変数の型集合は展開せずに解析を行う。これにより、生成される型集合の複雑さを、プログラム中に項として直接記述されている構造型データの複雑さ程度に抑えることができる。また、再帰構造があっても無限長の型が生成されなくなるため、解析が終了しなくなるのを防ぐことができる。

* 型集合中の型不明要素を表す。

次項で述べる型解析アルゴリズムでは、最初にすべての変数の型集合を $\{*\}$ とおき、組込述語や同一化による制約から、徐々に型集合の要素を決定していく。ただし、入出力関係など引数の型が決まらない組込述語が存在する場合や、プログラムの部分解析を行う場合など、最終的に得られる型集合が '*' を含むこともある。

以上の定義より、たとえばボディゴール `add(Y,1,Z),X=f(Z)` に対し、$T(\mathtt{Z}) = \{i\}$, $T(\mathtt{X}) = \{s(\mathtt{f}, 1, \{Z\})\}$ となる。

本手法で変数の型を型集合で表現しているのは、Mishra らの型解析 [35] の、取り得る型の結合で表現する方法と同様な考え方である。また、プログラム中に出現する構造型データをそのまま型として利用し、引数として現れる変数を型変数に置き換える手法は、Zobel の型解析 [32] でも用いられている。

3.3.3 型解析

続いて、この型情報の解析アルゴリズムを説明する。まず、型集合に対し、以下の演算と関数を定義する。なお、本項では、組込述語は=/2 だけを考える。他の組込述語に対する解析については、3.3.4 項で述べる。

型集合の OR 演算

$T_{OR} = OR(T_A, T_B)$ とすると、T_{OR} は以下を満たす最小の集合である。

1. アトミックデータ型 t_a について
   $(T_A \ni t_a) \vee (T_B \ni t_a)$

   $\rightarrow T_{OR} \ni t_a$

2. 構造型データ型 $t_{f/n}$ について
   $(T_A \ni s(f, n, (T_{A,1}, \ldots, T_{A,n}))) \wedge (T_B \ni s(f, n, (T_{B,1}, \ldots, T_{B,n})))$

   $\rightarrow T_{OR,k} = OR(T_{A,k}, T_{B,k})$ とおくと $T_{OR} \ni s(f, n, (T_{OR,1}, \ldots, T_{OR,n}))$

   $(T_A \ni t_{f/n} \wedge T_B \not\ni t_{f/n}) \vee (T_A \not\ni t_{f/n} \wedge T_B \ni t_{f/n})$

   $\rightarrow T_{OR} \ni t_{f/n}$

3. $(T_A \ni v_j) \vee (T_B \ni v_j)$

$\rightarrow T_{OR} \ni v_j$

4. $(T_A \ni *) \vee (T_B \ni *) \rightarrow T_{OR} \ni *$

この演算は、意味的には2個の型集合の和集合を求めるものである。ただし、簡潔な型集合を得るために、構造型データ型については名前と引数個数が一致するものをまとめている。

また、以下の記述では、$OR(...OR(OR(T_1,T_2),T_3),...,T_n)$ を $OR(T_1,...,T_n)$ と書く。

型集合の同一化

本研究の型解析手法では、各種の組込述語や同一化によって課せられる型制約より、各変数が取り得る型の集合を求める。二つの変数 $\mathrm{v}_A, \mathrm{v}_B$ がプログラム中で同一化される場合、それらの型集合 $T(\mathrm{v}_A), T(\mathrm{v}_B)$ は、お互いに相手と矛盾しないような集合 T_{UNIFY} に制約される。本論文では、これを型集合の同一化制約と呼び、制約された集合で $T(\mathrm{v}_A), T(\mathrm{v}_B)$ を更新することを、型集合の同一化と呼ぶ。

ここで、同一化制約の結果、得られる集合 T_{UNIFY} について考察する。$T(\mathrm{v}_A)$ にしか含まれない型は、$T(\mathrm{v}_B)$ と矛盾するため、T_{UNIFY} には含まれない。したがって直感的には、T_{UNIFY} は $T(\mathrm{v}_A)$, $T(\mathrm{v}_B)$ の積集合となる。しかし、$T(\mathrm{v}_B)$ に不明要素 '*' や型変数 v_j が含まれる場合、$T(\mathrm{v}_A)$ の任意の要素はこれと対応し得るので、$T(\mathrm{v}_A)$ にしか含まれない型も T_{UNIFY} に含まれる。たとえば、$T(\mathrm{v}_A) = \{a, i\}, T(\mathrm{v}_B) = \{i, f\}$ の場合、$T_{UNIFY} = \{i\}$ となるが、$T(\mathrm{v}_A) = \{a, i\}, T(\mathrm{v}_B) = \{i, f, *\}$ の場合、$T_{UNIFY} = \{a, i\}$ となる。

さらに、$T(\mathrm{v}_A)$ に構造型データ型が含まれ、その引数として型変数 v_k が現れる場合を考える。このとき、$T(\mathrm{v}_k)$ も $T(\mathrm{v}_B)$ の対応する要素により、同一化制約を受ける。たとえば、$T(\mathrm{v}_A) = \{s(f, 1, (\{v_i\}))\}, T(\mathrm{v}_B) = \{s(f, 1, (\{i, f\}))\}$ の場合、$T(\mathrm{v}_i) \ni i, f$ となる。ただし、このような同一化は実行時に v_A がこの構造型データ型の値をもつかどうかに依存するから、決定的なものではない。上記の例で言うと、$T(\mathrm{v}_i)$ が型 i, f をとる可能性があるというだけであり、$T(\mathrm{v}_i)$ に関する他の制約について、$T(\mathrm{v}_i) = \{i\}$ のようなより強い制約を緩和することも、$T(\mathrm{v}_i) = \{a\}$ のような異なる型への制約を否定することもない。すなわち、$T(\mathrm{v}_i)$ に不明要素や型変数が存在する場合のみ $\{i, f\}$ を追加できる。そこで、$T(\mathrm{v}_i) \leftarrow \{i, f\}$ とする代わりに、$T(\mathrm{v}_i) \leftarrow unify(T(\mathrm{v}_i), \{i, f\} \cup \{*\})$ とする。

以上の考察より、同一化制約の結果となる型集合 T_{UNIFY} を返す関数[1] $unify(T_A, T_B)$ を、図 3.1 のアルゴリズムで定義する。このアルゴリズムでは、まず 1. で結果 T_{UNIFY} の初期値を空集合とし、2. で T_A に含まれる型 t_i について、t_i が T_B にも含まれるか T_B が不明要素や型変数を含んでいれば、T_{UNIFY} にも含まれるとする。また、構造型データ型については、各引数にこの関数を再帰的に適用する。続いて、3. で T_A に含まれる型変数 v_i について、T_B が不明要素や型変数を含んでいれば、T_{UNIFY} にも含まれるとする。また、v_i が T_{UNIFY} に含まれるか否かに関わらず、プログラム変数 v_i は T_B に含まれる任意の型を取り得るので、T_B の各要素を $T(\mathrm{v}_i)$ に加える。4. で T_B の各型・型変数についても同様に処理し、最後に 5. で、不明要素 * が一方に含まれており、両型集合とも不明要素や型変数を含んでいるなら、T_{UNIFY} にも * が含まれるとする。

ここで定義した型集合に対する同一化は、Bansal らの研究 [36] で用いられている抽象項に対する抽象同一化と、概念的には同じものである。ただし、Bansal らの解析手法はトップダウンの抽象解釈を用いているため、抽象ゴールの実行に際して、開始状態に対して抽象同一化の結果を適用し、それを終了状態としている。一方、本研究では、時系列を考えない制約充足問題として解

[1] 型変数にも再帰的に制約を適用するという副作用を持つため、この関数は純関数ではない。

1. $T_{UNIFY} \leftarrow \emptyset$ /* 返り値の初期化 */
2. forall $t_i \in T_A$ {
   if t_iがアトミックデータ型 {
   if $t_i \in T_B \vee * \in T_B \vee v_j \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{t_i\}$
   }
   }
   else if $t_i = s(f, n, (T_{A,1}, \ldots, T_{A,n}))$ { /* 構造型データ型 */
   if $* \in T_B \vee v_j \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{t_i\}$
   }
   else if $s(f, n, (T_{B,1}, \ldots, T_{B,n})) \in T_B$ {
   $T_{UNIFY} \leftarrow OR(T_{UNIFY}, \{s(f, n, (unify(T_{A,1}, T_{B,1}), \ldots, unify(T_{A,n}, T_{B,n})))\})$
   }
   }
   }
3. forall $v_i \in T_A$ {
   if $* \in T_B \vee v_j \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{v_i\}$
   }
   $T(\mathrm{v}_i) \leftarrow unify(T(\mathrm{v}_i), T_B \cup \{*\})$
   }
4. T_Bについても、2,3 と同様に処理
5. if $(* \in T_A \wedge (* \in T_B \vee v_j \in T_B)) \vee (* \in T_B \wedge (* \in T_A \vee v_j \in T_A))$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{*\}$
   }
6. $T_{UNIFY} = \emptyset$ なら同一化失敗、そうでなければT_{UNIFY}を返す

図 3.1: 関数 $unify(T_A, T_B)$ のアルゴリズム

析を行っている。このため、次に述べる型解析アルゴリズムに示されているように、型集合間の同一化により生じる制約は、そのまま各型集合に対する制約として用いられる。

以上の演算と関数を用いて、次に型解析アルゴリズムを述べる。

型解析アルゴリズム

プログラム中の全ゴールの集合 G, 全変数の集合 Vが与えられたとき、各 $\mathrm{v}_i \in V$の型集合 $T(\mathrm{v}_i)$は、図 3.2のアルゴリズムで求めることができる。$H_{p/n,m}$と $B_{p/n,m}$はそれぞれクローズヘッド p/nとボディゴール p/n の第 m 引数の集合を表し、集合 Nは型集合が更新された変数のプールとして機能する。また、具体値 b_iの型を $t(b_i)$ で表している。

本アルゴリズムでは、まず 1. で更新プール Nを空にし、2. で各変数の型集合の初期値をすべて不明 ($\{*\}$) とする。続いて、3. で具体値との同一化による型制約を求め、4. でクローズヘッドやボディゴールの引数として出現する具体値を、更新プールに入れる。

続いて、5. でこれらの型制約の伝搬を解析する。まず、(a) では、節内の制約伝搬を追跡する。=/2 により両辺の変数に掛かる型制約は決定的であるため、関数 $unify$ により型集合間の同一化を行う。続いて (b), (c) では、節間の制約伝搬を追跡する。この伝搬は、ゴールのリダクション時に

1. $N \leftarrow \emptyset$ /* 更新変数集合を初期化 */
2. forall $v_i \in V$ {
   $T(\mathrm{v}_i) \leftarrow \{*\}$ /* 各変数の型集合を初期化 */
   }
3. forall $=(\mathrm{v}_j, b_k) \in G$ {
   $T(\mathrm{v}_j) \leftarrow \{t(b_k)\}$, $N \leftarrow N \cup \{\mathrm{v}_j\}$
   }
4. forall $b_h \in H_{p/n,m}$, $b_b \in B_{p/n,m}$ {
   $N \leftarrow N \cup \{b_h\}$, $N \leftarrow N \cup \{b_b\}$
   }
5. while $N \neq \emptyset$ {
   $x \leftarrow N$の要素, $N \leftarrow N \setminus \{x\}$

(a) if $x \in V$ {
   forall $=(x, \mathrm{v}_k) \in G$ {
   $T_{UNIFY} \leftarrow unify(T(x), T(\mathrm{v}_k))$, $T(x) \leftarrow T_{UNIFY}$, $T(\mathrm{v}_k) \leftarrow T_{UNIFY}$
   変更のあった各$T(\mathrm{v}_u)$について、それぞれ$N \leftarrow N \cup \{\mathrm{v}_u\}$
   }
   }

(b) forall $H_{p/n,m}$ $s.t.$ $x \in H_{p/n,m}$ {
   $T_{HB} \leftarrow \emptyset$
   forall $\mathrm{v}_h, b_h \in H_{p/n,m}$ {
   $T_{HB} \leftarrow OR(T_{HB}, T(\mathrm{v}_h))$, $T_{HB} \leftarrow OR(T_{HB}, \{t(b_h)\})$
   }
   forall $\mathrm{v}_b \in B_{p/n,m}$ {
   $T_{UNIFY} \leftarrow unify(T(\mathrm{v}_b), T_{HB})$, $T(\mathrm{v}_b) \leftarrow T_{UNIFY}$
   変更のあった各$T(\mathrm{v}_u)$について、それぞれ$N \leftarrow N \cup \{\mathrm{v}_u\}$
   }
   }

(c) forall $B_{p/n,m}$ $s.t.$ $x \in B_{p/n,m}$ {
   $T_{BH} \leftarrow \emptyset$
   forall $\mathrm{v}_b, b_b \in B_{p/n,m}$ {
   $T_{BH} \leftarrow OR(T_{BH}, T(\mathrm{v}_b))$, $T_{BH} \leftarrow OR(T_{BH}, \{t(b_b)\})$
   }
   forall $\mathrm{v}_h \in H_{p/n,m}$ {
   $T_{UNIFY} \leftarrow unify(T(\mathrm{v}_h), T_{BH})$, $T(\mathrm{v}_h) \leftarrow T_{UNIFY}$
   変更のあった各$T(\mathrm{v}_u)$について、それぞれ$N \leftarrow N \cup \{\mathrm{v}_u\}$
   }
   }

(d) forall $v_k \in V$ {
   if $T(\mathrm{v}_k) \ni v_k$ {
   $T(\mathrm{v}_k)$からv_kを除く
   }
   }
   }

図 3.2: 型解析アルゴリズム

クローズヘッド・ボディゴール間で引数が同一化されることに対応する。実行時にどのクローズヘッドが選ばれるかは非決定的であるため、ボディゴール p/n の引数である変数 v_bは、各クローズヘッド p/n の対応する位置の引数のいずれでも取り得る。したがって、(b) v_bに掛かる型制約としては、全クローズヘッド p/n の対応する位置の引数の型集合を OR 演算で融合したものになる。同様に、クローズヘッド p/n はすべてのボディゴール p/n から呼ばれる可能性があるから、(c) その引数である変数 v_hに掛かる型制約としては、全ボディゴール p/n の対応する位置の引数の型集合を OR 演算で融合したものになる。また、(d) 型集合の同一化の結果、$T(v_j)$ 自身に v_jが含まれる場合があるから、これを取り除く。

この解析の途中で、型集合間の同一化に失敗した場合は、対応するプログラム中の同一化は必ず失敗すると言える。したがって、そのプログラムは誤りを含んでいる。

アルゴリズムの停止性

ここでは、前記のアルゴリズムが有限時間で解析を終了して停止することを示す。

あるプログラム Pに対して、考えられるすべての型集合のうち、型だけからなるものの集合を $S = \{T_1, \ldots, T_m\}$、不明要素や型変数を含むものの集合を $S' = \{T_{m+1}, \ldots, T_n\}$ とする。P内に出現する型や変数は有限であるから、Sおよび S'は有限集合である。また、すべての型を含む型集合を $T_{Smax} \in S$, すべての型、型変数、および$*$を含む型集合を $T_{S'min} \in S'$とする。

ここで、型集合間の包含関係を次のように定義する。

型集合間の包含関係 2つの型集合 T_A, T_Bについて、$OR(T_A, T_B) = T_A$が成立するとき、$T_A \supseteq T_B$である。

たとえば、$T_A = \{a, i, s(f, 1, (\{*, i\}))\}, T_B = \{a, s(f, 1, \{i\})\}, T_C = \{a, i\}$ について、

$$
\begin{aligned}
OR(T_A, T_B) &= \{a, i, s(f, 1, (\{*, i\}))\} = T_A \\
OR(T_A, T_C) &= \{a, i, s(f, 1, (\{*, i\}))\} = T_A \\
OR(T_B, T_C) &= \{a, i, s(f, 1, (\{i\}))\}
\end{aligned}
$$

であるので、$T_A \supseteq T_B, T_A \supseteq T_C, T_B \not\supseteq T_C, T_C \not\supseteq T_B$である。

次に、Sについて以下のように半順序関係を定義する。

1. 任意の $T_a \in S$について、$\emptyset \leq T_a$
2. 任意の $T_a \in S$について、$T_a \leq T_{Smax}$
3. $T_a \subseteq T_b$なら、$T_a \leq T_b$

同様に、S'についても以下のように半順序関係を定義する。

1. 任意の $T'_a \in S'$について、$T_{S'min} \leq T'_a$
2. $T'_a \supseteq T'_b$なら、$T'_a \leq T'_b$

また、任意の $T \in S, T' \in S'$について、$T \leq T'$とする。
以上の定義によって、S, S'は半順序集合となる。

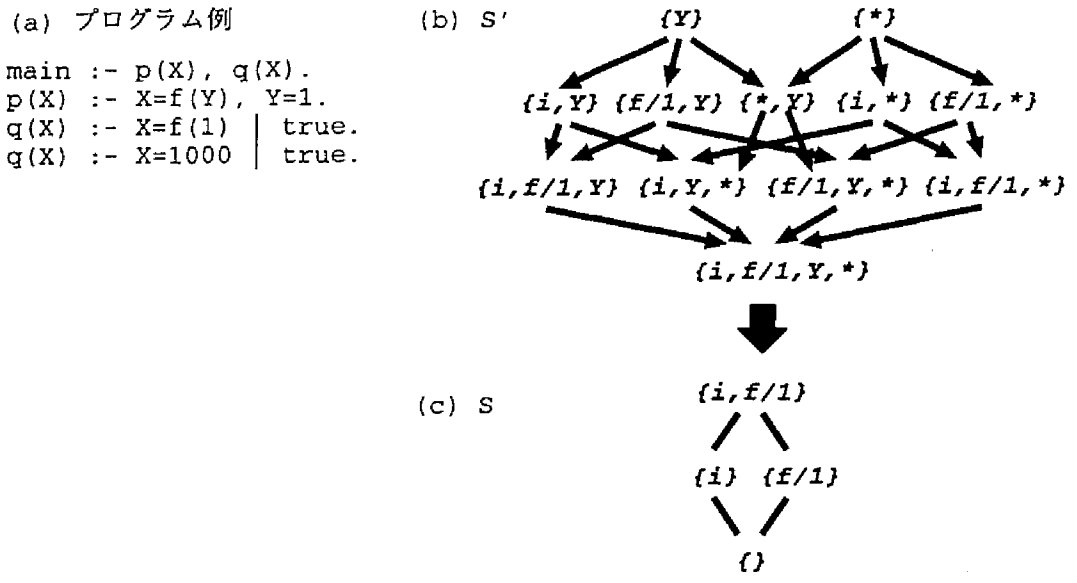


図 3.3: 型集合の半順序集合

たとえば、図 3.3(a) のプログラムでは、プログラム中に出現する型は $i, t_{f/1}$、型変数は Y であるから、S', Sはそれぞれ図 3.3(b), 3.3(c) のようになる[2]。

直感的には、$T_a \leq T_b$なら T_aは T_bよりも曖昧性が少ないことを意味する。もっとも曖昧なのは、型集合の要素がまったく不明のとき ($\{*\}$) や型変数のみの場合であり、もっとも曖昧でないのは、型集合が要素をまったく持たないとき ($\emptyset$) である。型集合が不明要素 '$*$' を含むときは、その他の判明した要素が多ければ、それだけ曖昧性が少ない。また、型集合が不明要素 '$*$' を含まないときは、この集合に含まれない型は取り得ないことが確定しているから、集合の要素が少ないほど、取り得る型が絞り込まれた、すなわち曖昧性が減少したことを意味している。

本手法の型解析アルゴリズムは、制約により曖昧性を減少させていき、それ以上絞り込めなくなった時点で終了する。具体的には、図 3.2のアルゴリズムで、2. および 3. で型集合の初期値を与えた後、5. のループ内で各型集合を更新する際には、必ず関数 *unify* の結果を用いる。そして、図 3.1の定義より、$unify(T_A, T_B) \leq T_A, T_B$である。したがって、解析の過程で各型集合を更新する操作 $T_{old} \rightarrow T_{new}$ に対して、必ず $T_{new} \leq T_{old}$となる。また、変数プール Nへの追加は、型集合が更新された場合のみ行う。よって、本アルゴリズムでは、有限時間ですべての変数の型集合が制約より得られる下界に到達し、Nが空になって解析が終了する。

アルゴリズムの正しさ

3.2 節で述べたように、一般にプログラムの静的解析では正確な情報を得ることは難しい。しかし、少なくとも誤りではないことを保証しておく必要がある。

本研究の解析では、型集合によって、各変数が取る可能性のある型の範囲を表現している。したがって、実際に取り得る型を含まない型集合は誤っているが、逆に実際には取り得ない型を含んでいる場合は、その型集合は正確 (accurate) ではないものの正しい (correct) といえる。

[2] $t_{f/1}$の引数の型集合については省略してあるが、これについても再帰的に同様な半順序関係を考えれば良い。

(a) ゴール引数間の関係

```
main :- p(X,Y), ....
p(X,Y) :- X=a | Y=b.
p(X,Y) :- X=1 | Y=2.
```

(b) 実行されない節による制約

```
main :- p(a,Y), ....
p(X,Y) :- X=a | Y=b.
p(X,Y) :- X=b | Y=2.
```

図 3.4: 完全な解析情報が得られない例

図 3.2の解析アルゴリズムでは、型集合の初期値は型不明要素、すなわちすべての型を取り得るとしている。したがって、この段階では各型集合は正しい。続いて組込述語や同一化による型制約を適用し、型の範囲を狭めていく。このときこれらの型制約が、実際には取り得る型を除外してしまうことがなければ、制約適用後の型集合も正しいことが保証できる。以下、このような型制約を正しい型制約と呼ぶ。

組込述語による制約は、述語の定義により正しい型制約を与えることができる。同一化述語による型制約は、関数 $unify(T_A, T_B)$ の定義より両型集合に矛盾しない型すべてを含むから、正しい制約である。また、クローズヘッド・ボディゴール間の同一化では、OR 演算によって各プログラムパスで出現し得る型すべてを含むような制約が掛けられる。したがって、この型制約も正しい。

以上、解析アルゴリズム中の型制約はすべて正しいから、最終的に正しい型集合が得られる。

アルゴリズムの正確さ

次に、この解析手法の結果が、どれほど正確であるかについて考察する。

論理型言語の解析では、同一化によって複数の変数が同じ値を取る別名化 (aliasing) が、精度上の問題を引き起こす。たとえば、X=Y という同一化が起きた場合、X に関する型制約は Y にも適用されなければならない。別名化は構造型データの引数として現れる変数についても発生し、たとえば二つの同一化ゴール X=f(Y)、W=f(Z) がプログラム中の離れた箇所に存在する場合でも、X と W の同一化が起これば、Y と Z も互いの別名変数となる。このため、実行の各段階における状態を求めていく抽象解釈では、この別名化を正しく解析するための工夫が必要となる。

本研究の手法では、時刻に依存しない大域的な制約を求めるため、個々の時点での状況を考える必要はなく、生じ得る別名化がすべて結果に反映されていることを保証すれば良い。まず、ゴールの引数として直接現れる変数間の同一化については、一方の変数の型集合が更新されると、再び同じゴールを通した制約伝搬が発生し、もう一方の変数にも必ず同じ制約が伝わる。したがって、別名化は正しく処理されている。また、本手法では実際のプログラム中の項や引数として出現する変数を、そのまま型や型変数に対応させている。このため、構造型データの引数として現れる変数に関する別名化は、型集合間の同一化において型変数間の別名化として捕えることができる。関数 *unify* のアルゴリズムより、同一化された型変数は互いに相手の型集合に加えられるため、最終的に得られた型集合には、別名化された変数の型集合の要素が反映される。

一方、本手法で得られた型情報は、実際には生じ得ない型を含んでしまう可能性がある。

まず、本手法の結果は各変数の型集合が独立して得られるから、変数間に取り得る型に関する何らかの関係があっても、それに関する情報は得られない。たとえば、図 3.4(a) では、ゴール p/2 の 2 引数はともに記号アトム、あるいはともに整数になるが、本手法の結果は $T(\mathrm{X}) = \{a, i\}, T(\mathrm{Y}) = \{a, i\}$ という形になるため、実際には有り得ない型の組み合わせも含んでしまう。

また、実際の値によっては分岐しない節による制約も、無条件に採用してしまう。たとえば、

$$\texttt{arg(P,f(1,a,[]),A)} \rightarrow T(\texttt{A}) = \{i, a\}$$
$$\texttt{arg(2,f(1,a,[]),A)} \rightarrow T(\texttt{A}) = \{a\}$$

図 3.5: 定数引数の利用

図 3.4(b) では、ゴール p/2 の第 1 引数の値が a であるため、実行時には p/2 の定義節として必ず上の節が選ばれ、Y の値の型は必ず a となる。しかし、本手法の結果は $T(\texttt{X}) = \{a\}, T(\texttt{Y}) = \{a, i\}$ と、実際には取り得ない型を含んでしまう。

3.3.4 組込述語の扱いと型情報の拡張

前節までで述べてきた型情報および解析アルゴリズムでは、実際の KL1 プログラムに出現する様々な組込述語を考慮していない。基本的には、各組込述語には入出力引数の型が決められているため、それにより生じる制約を、同一化述語による制約と同様に適用していけばよい。しかし、一部の組込述語には、出力の型が入力の型や値に依存するものや、任意のファンクタといった本手法の型情報では表現できない制約が掛かるものがある。

本節では、こうした組込述語の扱いについて考察し、3.3.2節で定義した型情報に、いくつかの拡張を加える。

定数引数の利用

組込述語には、入力引数の値が出力値の型に影響を与えるものがある。たとえば 2.2.4 項で述べたように、arg(P,F,A) は、ファンクタ F の P 番目の引数と A を同一化する。ここで、$T(\texttt{F}) = \{s(f, n, (T_1, \ldots, T_n))\}$ であるとすると、P が実行時まで決まらない場合には、A が F のどの引数になるかは静的に決定できない。よってこの組込述語による制約は、$T(\texttt{A}) \leftarrow \mathit{unify}(T(\texttt{A}), OR(T_1, \ldots, T_n))$ となる。しかし、プログラム中で直接引数に値が書かれている場合は、さらに型を絞り込むことができる。たとえば、arg(2,F,A) と書かれていれば、$T(\texttt{A}) \leftarrow \mathit{unify}(T(\texttt{A}), T_2)$ とできる。

そこで、本解析手法では、組込述語の引数に具体値が直接書かれている場合は、型制約の決定にその値を利用することにする (図 3.5)。

出力型が入力型に依存する組込述語

組込述語には、出力引数の型が入力引数の型に依存するものがある。たとえば、arg(P,F,A) は、上で述べたように $T(\texttt{F})$ が決まっていれば、$T(\texttt{A})$ に関する型制約を与えることができる。しかし、一般には F は他のゴールにより具体化されるから、F の具体化ゴールから $T(\texttt{F})$ が伝搬してくるまでは、$T(\texttt{A})$ の型制約を求めることができない。

そこで、この種の組込述語に関しては、図 3.2の型解析アルゴリズムで、3. の組込述語による型集合決定の際には $T(\texttt{A})$ を生成しない。そして、5. の制約伝搬追跡ループで、(a) で=/2 による伝搬を求めるのと同様に、Nから変数 Fが取り出されたら $T(\texttt{F})$ より $T(A)$ への制約を求め、$T(A)$ を更新する。また、他の制約により $T(\texttt{F})$ がさらに絞り込まれることがあるが、この場合は $T(\texttt{F})$ が更新されて Nに F が再び入るので、同様に $T(A)$ への制約を求め直して、$T(A)$ を更新する (図 3.6)。

```
1 main     :- p(F,X,Y), q(F,Y), r(F).
2 p(F,X,Y) :- Y = 1 | arg(2,F,X).
3 q(F,Y)   :- F = f(1,2,3) | Y = 1.
4 q(F,Y)   :- F = g(a,b)   | Y = 2.
5 r(F)     :- F = f(1,2,3).
```

節iの変数をX^iの形で表すと、
初期値：$T(F^2) = \{*\}$ より $T(X^2) = \{*\}$
$\Rightarrow T(F^1) \leftarrow OR(T(F^3), T(F^4)) = \{s(f,3,(\{i\},\{i\},\{i\})), s(g,2,(\{a\},\{a\}))\}$
$\Rightarrow T(F^2) \leftarrow T(F^1) = \{s(f,3,(\{i\},\{i\},\{i\})), s(g,2,(\{a\},\{a\}))\}$
⇒arg/3 より $T(X^2) \leftarrow \{a,i\}$
$\Rightarrow T(F^1) \leftarrow T(F^5) = \{s(f,3,(\{i\},\{i\},\{i\}))\}$
$\Rightarrow T(F^2) \leftarrow T(F^1) = \{s(f,3,(\{i\},\{i\},\{i\}))\}$
⇒arg/3 より $T(X^2) \leftarrow \{i\}$

図 3.6: 入力引数の型集合による出力引数の型集合の更新

構造型データ型の拡張

同一化述語=/2による変数の具体化では、具体値がプログラム中に直接記述される。このため、構造型データ型についても、ゴール X=f(1,a,[]) に対して $T(X) = \{s(f,3,(\{i\},\{a\},\{a\}))\}$ のように、一意に決定することができる。

しかし、構造型データを出力とする組込述語には、その形が動的に決まるものがある。たとえば、2.2.4 項で述べたように、new_functor(F,N,A) は、F をファンクタ名が N で A 個の 0 を引数とするファンクタで具体化する。したがって、N や A の値が静的に判らない限り、F の構造型データ型を一意に表すことができない。new_functor(F,f,A) のように N が具体値で書かれていれば、$T(F) = \{a, s(f,1,(\{i\})), s(f,2,(\{i\},\{i\})), \ldots\}$ [3] のように、取り得る型を列記する形で表すこともできる。しかしこの場合、型集合が無限集合になるため、解析アルゴリズムの停止性が保証できなくなってしまう。

そこで、本手法では 3.3.2 項で定義した構造型データ型を拡張し、ファンクタ名や引数個数が定まっていないものを許す、拡張構造型データ型を導入する。

拡張構造型データ型の定義　拡張構造型データ型は、次のうちのいずれかである。

$$
\begin{array}{ll}
s(name, n, (T_1, \ldots, T_n)) & (1) \\
s(*, n, (T_1, \ldots, T_n)) & (2) \\
s(name, *, T_{args}) & (3) \\
s(*, *, T_{args}) & (4)
\end{array}
$$

3.3.2 項で定義した (1) の他に、この拡張では (2),(3),(4) のように、ファンクタ名や引数個数が不定のものを許し、*の部分は任意のファンクタ名や引数個数にマッチし得るとする。これらを抽

[3] 組込述語 new_functor/3 の第 3 引数は 0 以上の整数をとり、引数個数 0 のファンクタは、ファンクタ名と同名の記号アトムを意味する。

象構造型データ型と呼ぶ。また、引数個数が*の場合、引数の型集合はT_{args}が一つだけ与えられる。これは、この型に対応する実際の構造型データの引数が、どれもT_{args}の要素のいずれかの型を取り得ることを表している。

たとえば、$s(*,2,(\{i\},\{i\}))$は任意の名前で2個の整数引数を持つファンクタを表すので、`f(1,2)`や`g(0,0)`にマッチする。しかし、引数個数の異なる`f(1,2,3)`や、引数の型が異なる`g(a,a)`にはマッチしない。同様に、$s(\mathtt{f},*,\{i,a\})$は名前が`f`で、任意個数の引数を持つファンクタを表す。各引数が取り得る型は整数または記号アトムである。したがって、この型は`f(1,2)`や`f(0,a,1,b)`、さらに記号アトム`f`にマッチする。しかし、ファンクタ名の異なる`g(1,2,3)`や、引数の型が異なる`f(a,g(1))`にはマッチしない。

以下、3.3.3 項で定義した型解析を拡張し、この抽象構造型データ型を扱えるようにする。なお、前記したように名前f、引数個数nである構造型データ型を$t_{f/n}$と表記し、同様に名前や引数個数が不定の場合を$t_{*/n}$, $t_{f/*}$, $t_{*/*}$のように書く。

型集合のOR演算の拡張 演算$OR(T_A,T_B)$は、意味的には2つの型集合T_A, T_Bの和集合を求めるものである。したがって、次の例のように、不定要素を含まない構造型データ型を、これとマッチする不定要素を含んだ抽象構造型データ型に含めてしまっても、意味的には問題ない。

$$OR(\{s(\mathtt{f},1,(\{i\})\},\{s(*,1,(\{i\})\}) = \{s(*,1,(\{i\})\}$$
$$OR(\{s(\mathtt{f},2,(\{i\},\{a\})\},\{s(\mathtt{f},*,\{i,a\}\}) = \{s(\mathtt{f},*,(\{i,a\})\}$$

しかし、最終的に不定要素が確定しない場合には、一部の判明した構造型名や引数個数が不定要素に吸収されて消えてしまい、得られる解析情報の精度が落ちる可能性がある。そこで、本手法のOR演算では、抽象構造型データ型は、同じ位置に不定要素を含むもの同士のみを合わせて一つのデータ型とし、その他の組合せは、たとえマッチし得るものであっても別々に残すことにする。具体的には、3.3.3 項で述べたOR演算の定義のうち、構造型データ型に関する部分を、以下のように拡張する。

2. 構造型データ型について

(a) $(T_A \ni s(f,n,(T_{A,1},\ldots,T_{A,n}))) \wedge (T_B \ni s(f,n,(T_{B,1},\ldots,T_{B,n})))$
$\rightarrow T_{OR,k} = OR(T_{A,k},T_{B,k})$ とおくと $T_{OR} \ni s(f,n,(T_{OR,1},\ldots,T_{OR,n}))$
$(T_A \ni t_{f/n} \wedge T_B \not\ni t_{f/n}) \vee (T_A \not\ni t_{f/n} \wedge T_B \ni t_{f/n})$
$\rightarrow T_{OR} \ni t_{f/n}$

(b) $(T_A \ni s(*,n,(T_{A,1},\ldots,T_{A,n}))) \wedge (T_B \ni s(*,n,(T_{B,1},\ldots,T_{B,n})))$
$\rightarrow T_{OR,k} = OR(T_{A,k},T_{B,k})$ とおくと $T_{OR} \ni s(*,n,(T_{OR,1},\ldots,T_{OR,n}))$
$(T_A \ni t_{*/n} \wedge T_B \not\ni t_{*/n}) \vee (T_A \not\ni t_{*/n} \wedge T_B \ni t_{*/n})$
$\rightarrow T_{OR} \ni t_{*/n}$

(c) $(T_A \ni s(f,*,T_{Aargs})) \wedge (T_B \ni s(f,*,T_{Bargs}))$
$\rightarrow T_{OR} \ni s(f,*,OR(T_{Aargs},T_{Bargs}))$
$(T_A \ni t_{f/*} \wedge T_B \not\ni t_{f/*}) \vee (T_A \not\ni t_{f/*} \wedge T_B \ni t_{f/*})$
$\rightarrow T_{OR} \ni f/*$

(d) $(T_A \ni s(*,*,T_{Aargs}))\wedge(T_B \ni s(*,*,T_{Bargs}))$
$\rightarrow T_{OR} \ni s(*,*,OR(T_{Aargs},T_{Bargs}))$
$(T_A \ni t_{*/*} \wedge T_B \not\ni t_{*/*}) \vee (T_A \not\ni t_{*/*} \wedge T_B \ni t_{*/*})$
$\rightarrow T_{OR} \ni t_{*/*}$

この定義による OR 演算の例を、以下に示す。

$$OR(\{s(\mathtt{f},1,(\{i\}))\},\{s(*,1,(\{i\}))\}) = \{s(\mathtt{f},1,(\{i\}),s(*,1,(\{i\}))\}$$
$$OR(\{s(\mathtt{f},2,(\{i\},\{a\}))\},\{s(\mathtt{f},*,\{i,a\}\}) = \{s(\mathtt{f},2,(\{i\},\{a\}),s(\mathtt{f},*,(\{i,a\}))\}$$
$$OR(\{s(*,*,\{i,a\})\},\{s(*,*,\{*,f\}\}) = \{s(*,*,\{*,i,a,f\})\}$$

型集合の同一化の拡張 3.3.3 項で述べたように、型集合の同一化制約 T_{UNIFY} は、意味的には 2 つの型集合 T_A, T_B の双方に矛盾しない型集合である。したがって OR 演算とは逆に、不定要素を含まない構造型データ型 f/n と、これとマッチする抽象構造型データ型 $*/n, f/*, */*$ が同一化されると、より曖昧性の高い後者は、前者によって制約される。同様に、抽象構造型データ型 $*/*$ は、より曖昧性の低い抽象構造型データ型 $*/n, f/*$ によって制約される。

また、抽象構造型データ型の導入によって、$t_{f/n} \in T_A$ に対し、T_B はこれとマッチし得る構造型データ型 $t_i,\ldots,t_j$ を複数含む可能性がある。この場合、$t_{f/n}$ の引数の型は、$t_i,\ldots,t_j$ のいずれかの、対応する引数の型を取り得る。このため、$t_{f/n}$ の引数に掛かる型制約は、$t_i,\ldots,t_j$ の対応する引数の型集合の OR になる。たとえば、$T_A = \{s(\mathtt{f},1,(\{a,i,f\}))\}$, $T_B = \{s(\mathtt{f},1,(\{a\})),s(*,1,(\{i\}))\}$ に対して、$t_{\mathtt{f}/1} \in T_A$ は $t_{\mathtt{f}/1}, t_{*/1} \in T_B$ の双方とマッチし得る。したがって、$t_{\mathtt{f}/1} \in T_A$ の第 1 引数が受ける制約は、$t_{\mathtt{f}/1}, t_{*/1} \in T_B$ の第 1 引数の型集合の OR、すなわち $OR(\{\mathrm{a}\},\{\mathrm{i}\})=\{\mathrm{a,i}\}$ である。よって、$unify(T_A,T_B) = \{s(\mathtt{f},1,(\{a,i\}))\}$ となる。

引数個数が不定の抽象構造型データ型 $t_{f/*}, t_{*/*}$ については、唯一の引数の型集合 T_{args} が、同一化相手のすべての引数に対する型制約となる。たとえば、$T_A = \{s(\mathtt{f},2,(\{a,i\},\{a,f\}))\}$, $T_B = \{s(\mathtt{f},2,(\{a\},\{f\})),s(*,*,\{i\}),\}$, に対して、$t_{\mathtt{f}/2} \in T_A$ の第 1、第 2 引数が受ける型制約はそれぞれ $OR(\{\mathrm{a}\},\{\mathrm{i}\})=\{\mathrm{a,i}\}$, $OR(\{\mathrm{f}\},\{\mathrm{i}\})=\{\mathrm{i,f}\}$ である。よって、$unify(T_A,T_B) = \{s(\mathtt{f},2,(\{a,i\},\{f\}))\}$ となる。

以上の結果を実現するよう、3.3.3 項で述べた関数 $unify(T_A,T_B)$ のアルゴリズムのうち、構造型データ型に関する部分を、図 3.7のように拡張する。

アルゴリズムの拡張 拡張した OR 演算と関数 *unify* を用いることにより、図 3.2の型解析アルゴリズムを変更することなく、抽象構造型データ型を含む型解析を行うことができる。

本項のような型情報の拡張を行っても、1 つのプログラム中に出現する抽象構造型データ型は有限個である。したがって、前に述べたアルゴリズムの停止性に関する議論は、拡張後の解析にもそのまま適用できる。実際、抽象構造型データ型は、`new_functor/3` のような一部の組込述語の引数について、通常の構造型データ型の代わりに出現するにすぎないので、プログラム全体を通して出現するデータ型の種類は、増加するわけではない。

拡張構造型データ型を用いた組込述語の処理 拡張構造型データ型を用いると、動的な構造型データを扱う組込述語を、次のようにして静的解析で扱うことができる。

1. 動的な構造型データを出力する組込述語

```
else if t_i = t_{f/n} ∨ t_i = t_{*/n} ∨ t_i = t_{f/*} ∨ t_i = t_{*/*} {   /* 構造型データ型 */
  if * ∈ T_B ∨ v_j ∈ T_B{
    T_UNIFY ← T_UNIFY ∪ {t_i}
  }
  else {
    T_{UNIFY,1} ← ∅, ..., T_{UNIFY,n} ← ∅
    if t_i = t_{f/n} ∧ s(f, n, (T_{B,1}, ..., T_{B,n})) ∈ T_B{
      T_{UNIFY,1} ← OR(T_{UNIFY,1}, T_{B,1}), ..., T_{UNIFY,n} ← OR(T_{UNIFY,n}, T_{B,n})
    }
    if (t_i = t_{f/n} ∨ t_i = t_{*/n}) ∧ s(*, n, (T_{B,1}, ..., T_{B,n})) ∈ T_B{
      T_{UNIFY,1} ← OR(T_{UNIFY,1}, T_{B,1}), ..., T_{UNIFY,n} ← OR(T_{UNIFY,n}, T_{B,n})
    }
    if (t_i = t_{f/n} ∨ t_i = t_{f/*}) ∧ s(f, *, T_{Bargs}) ∈ T_B{
      T_{UNIFY,1} ← OR(T_{UNIFY,1}, T_{Bargs}), ..., T_{UNIFY,n} ← OR(T_{UNIFY,n}, T_{Bargs})
    }
    if (t_i = t_{f/n} ∨ t_i = t_{f/*} ∨ t_i = t_{*/n} ∨ t_i = t_{*/*}) ∧ s(*, *, T_{Bargs}) ∈ T_B{
      T_{UNIFY,1} ← OR(T_{UNIFY,1}, T_{Bargs}), ..., T_{UNIFY,n} ← OR(T_{UNIFY,n}, T_{Bargs})
    }
    if T_{UNIFY,1} ≠ ∅{
      if t_i = s(f, n, (T_{A,1}, ..., T_{A,n})){
        T_UNIFY ← OR(T_UNIFY, {s(f, n, (unify(T_{A,1}, T_{UNIFY,1}), ..., unify(T_{A,n}, T_{UNIFY,n})))})
      }
      if t_i = s(*, n, (T_{A,1}, ..., T_{A,n})){
        T_UNIFY ← OR(T_UNIFY, {s(*, n, (unify(T_{A,1}, T_{UNIFY,1}), ..., unify(T_{A,n}, T_{UNIFY,n})))})
      }
      if t_i = s(f, *, T_{Aargs}){
        T_UNIFY ← OR(T_UNIFY, {s(f, *, unify(T_{Aargs}, T_{UNIFY,1}))})
      }
      if t_i = s(*, *, T_{Aargs}){
        T_UNIFY ← OR(T_UNIFY, {s(*, *, unify(T_{Aargs}, T_{UNIFY,1}))})
      }
    }
  }
}
```

図 3.7: 関数 $unify(T_A, T_B)$ の拡張

$$
\begin{array}{lcl}
\texttt{new\_functor(F,f,A)} & \rightarrow & T(\texttt{F}) = \{s(\texttt{f}, *, \{i\}\} \\
\texttt{new\_functor(F,N,2)} & \rightarrow & T(\texttt{F}) = \{s(*, 2, (\{i\}, \{i\})\} \\
\texttt{new\_functor(F,N,A)} & \rightarrow & T(\texttt{F}) = \{s(*, *, \{i\}\}
\end{array}
$$

図 3.8: 組込述語 new_functor/3 の出力引数の型

$$
\begin{array}{l}
T(\texttt{F}) = \{s(\texttt{f}, 2, (\{a\}, \{i\})), s(\texttt{g}, 1, (\{*, f\}))\}, \\
\quad \texttt{arg(1,F,A)} \rightarrow T(\texttt{A}) = \{*, a, f\} \\
T(\texttt{F}) = \{s(*, 2, (\{a\}, \{i\})), s(\texttt{g}, *, \{*, f\}\}, \\
\quad \texttt{arg(2,F,A)} \rightarrow T(\texttt{A}) = \{*, i, f\} \\
T(\texttt{F}) = \{s((*, 2, (\{a\}, \{i\})), s(*, *, \{*, f\}\}, \\
\quad \texttt{arg(P,F,A)} \rightarrow T(\texttt{A}) = \{*, a, i, f\}
\end{array}
$$

図 3.9: 組込述語 arg/3 の出力引数の型

出力データ構造を決める入力引数のうち、定数引数を取らない部分については静的に決定できないので、*を用いて不定とする。

たとえば、前述の new_functor(F,N,A) の場合、N, A が定数引数でなければ、それぞれファンクタ名、引数個数を*とした拡張構造型データ型で、F を制約する。したがって、出力引数の型集合の初期値は、図 3.8のようになる。

2. 動的な構造型データを入力とする組込述語

入力引数の型として拡張構造型データ型が与えられるので、それに応じて出力引数の型を制約する。

たとえば、前述の arg(P,F,A) の場合、F の判明している構造型データ型のうち、P 番目の引数の型で A を制約する。P が定数引数で与えられない場合は、A が何番目の引数で具体化されるか不明なため、全引数の型集合の OR で A を制約する (図 3.9)。

同様に、2.2.4 項で述べた、ファンクタ F の第 P 引数を E に置き換えたものと、NF を同一化する組込述語 setarg(P,F,E,NF) についても、得られた F の構造型データ型より NF の型制約を生成する。ただし、P が定数引数で与えられない場合は、置換される引数が一意に定まらないから、E の型集合をすべての引数の型集合に OR で加える。また、引数個数が不定の場合は、引数の型集合に E の型集合を OR で加える (図 3.10)。

大規模な構造型定数の簡略化

プログラム中に大規模な構造型定数が記述されている場合、これを単純に型に直すと、解析量が極端に多くなってしまう場合がある。この問題は、本項で導入した抽象構造型データ型を用いて型を簡略化することによって、回避することができる。

$$T(\mathtt{F}) = \{s(\mathtt{f}, 2, (\{a\}, \{i\}))\}, T(\mathtt{E}) = \{*, f\}$$
$$\mathtt{setarg(1,F,E,NF)} \rightarrow T(\mathtt{NF}) = \{s(\mathtt{f}, 2, (\{*, f\}, \{i\}))\}$$
$$T(\mathtt{F}) = \{s(*, 2, (\{a\}, \{i\}))\}, T(\mathtt{E}) = \{*, f\}$$
$$\mathtt{setarg(P,F,E,NF)} \rightarrow T(\mathtt{NF}) = \{s(*, 2, (\{*, a, f\}, \{*, i, f\}))\}$$
$$T(\mathtt{F}) = \{s(*, *, \{a, i\}\}, T(\mathtt{E}) = \{*, f\}$$
$$\mathtt{setarg(P,F,E,NF)} \rightarrow T(\mathtt{NF}) = \{s(*, *, \{*, a, i, f\})\}$$

図 3.10: 組込述語 setarg/4 の出力引数の型

たとえば、プログラム中で X=f(1,2,3,...,100) のようなゴールが記述されていた場合、単純に型集合を求めると、$T(\mathtt{X}) = \{s(\mathtt{f}, 100, (\{i\}, \ldots, \{i\}))\}$ となる。このように大規模な構造型データ型を生成してしまうと、型波及の追跡を行う際の OR 演算や同一化の処理量が大幅に増え、大量の解析時間が必要になる。

しかし実際には、このような大規模な構造型データは、一定の規則性を持っていることが多い。そこで本手法では、引数が一定以上の個数の構造型データ型は、たとえ引数個数が判明していても、引数個数が不定の抽象構造型データ型に簡略化して扱う。たとえば上記の例では、$T(\mathtt{X}) = \{s(\mathtt{f}, *, \{i\})\}$ とする。

この簡略化により解析の精度が落ちる可能性があるが、実際のプログラムで問題になることはほとんどないと考えられる。たとえば、f/100 のような大規模なファンクタに対しては、通常は同じ大きさのファンクタを直接ガード部に記述することはなく、組込述語 arg/3, setarg/4 などが使われる。また、これらの組込述語による操作も、f/100 の各引数を順に参照するなど、対象となる引数が動的にしか決まらないものが多い。したがって、上記のように抽象構造型データ型に変換してしまっても、精度にはほとんど影響を及ぼさないことが期待できる。

また、長いリストのような再帰構造についても、再帰の回数が一定以上なら型変数を用いて簡略化することで、解析量の削減を図ることができる。

たとえば、プログラム中で X=[1,2,...,100] のようなゴールが記述されていた場合、単純に型集合を求めると、$T(\mathtt{X}) = \{s(.,2,(\{i\},\{s(.,2,(\{i\},\{\ldots s(.,2,(\{i\},\{a\}))\ldots\})\}))\}$ となり、やはり解析時間の増加を引き起こす。そこで本手法では、構造型データへの具体化ゴール $\mathtt{v}_i = f(a_1, \ldots, a_n)$ について、f/n の中に同じ名前と引数個数の構造型データが一定回数以上入れ子になっている場合、次の手順で簡略化する。

1. $T_1 \leftarrow \emptyset, \ldots, T_n \leftarrow \emptyset$ とする。

2. 構造型データ f/n の各引数 a_j について、その型を $T(a_j)$ とすると、$T_j \leftarrow OR(T_j, T(a_j))$ とする。ただし、$a_j = f/n$ なら、$T_j \leftarrow OR(T_j, \{v_i\})$ とし、a_j について再帰的に処理を繰り返す。

3. 最終的に、$T(\mathtt{v}_i) = \{s(f, n, (T_1, \ldots, T_n))\}$ とする。

この結果、たとえば上記の例では、$T(\mathtt{X}) = \{s(.,2,(\{i\},\{X,a\}))\}$ となる。

ジェネリック・オブジェクト

KLIC ではベクタや浮動小数点数など、ジェネリック・オブジェクトで実装されたデータ型があるが、解析上はこうした実装方法の違いを考慮する必要はない。

ジェネリック・オブジェクトに関する操作は、一般にはメソッドを用いて行われる。メソッドは疑似述語で呼び出されるため、解析上は組込述語の一種として扱うことができる。しかし、どの種類のジェネリック・オブジェクトに対するメソッドであるかは、引数で指定されるため、静的解析では引数の型が決定できない場合がある。

たとえば、ジェネリック・オブジェクトの生成は

$$\texttt{generic:}new(\text{クラス名}, Object, Args, \ldots)$$

で行う。ここで、クラス名は記号アトムで与えるので、定数引数を取らない場合は、静的解析では一意に決定できない。したがって、この場合 *Object* は、ベクタや浮動小数点などジェネリック・オブジェクトで実装された型すべてを取り得るとせざるを得ない。

もっとも、ベクタなど標準で用意されているジェネリック・オブジェクトについては、多くのメソッドは同等の組込述語が用意されている。たとえば、ベクタの生成は組込述語 `new_vector/2` により行うことができる。こうした組込述語で行われる操作については、データ型自体の実装がジェネリック・オブジェクトであることを考慮せずに解析することができる。また、メソッド呼出による操作についても、前記した定数引数の利用を行うことによって、ある程度までオブジェクトの種類を限定できる。

ユーザが独自のジェネリック・オブジェクトを追加した場合は、それらに関する情報がコンパイル時に与えられない限り、静的解析は不可能である。したがって本研究の解析手法では、ユーザ定義のデータ型を使用しているプログラムについては、一部の変数がそのデータ型を取り得ることを検出できず、型集合の該当部分が型不明要素のまま残る可能性がある。

その他の KL1 の拡張機能

`alternatively`, `otherwise` は、節選択の順序に影響を与えるだけであるので、型解析の際には無視することができる。

また、`@node` や`@priority` などのプラグマについても、型制約に影響を与えることはない。

3.4 結言

以上、本章では論理型言語の静的解析手法について述べ、本研究で用いた KL1 の型解析手法について説明した。

KL1 に限らず、論理型言語の静的解析はプログラムの最適化など多くの利用価値があり、従来より様々な研究が行われてきた。本研究ではこれらの成果を踏まえ、実行最適化のための静的解析手法として、型解析を採用した。本解析手法では、型集合や型変数の導入によって、プログラムの非決定的な部分や再帰構造を伴う部分を、効率よく表現することができる。また、この型表現に対し、OR 演算と型集合の同一化を用いて制約充足を行い、各変数の型集合を求めるアルゴリズムを提案した。

また、KL1 のさまざまな組込述語が解析に与える影響について考察し、起こり得る問題を解決できるように型情報を拡張し、組込述語の処理方法について議論を行った。この結果、拡張された解析手法では、実プログラムに対する解析が可能であり、実用的なものになっている。

次の4,5章では、この型解析手法を利用した、KL1の実行最適化手法について述べる。

Chapter 4

通信の最適化

4.1 緒言

分散メモリ型並列計算機では、実行中に生じるプロセッサ間のメッセージ通信が、しばしばボトルネックとなる。とくに、KLIC など従来の KL1 の実装では、細粒度通信が頻発し、そのオーバヘッドが大幅な速度低下の要因になっている。本章では、この通信オーバヘッドを削減するためのメッセージ通信の最適化について述べる。

以下、4.2 節で従来の実装における問題点を述べ、4.3 節でこれを解決する手法の概要を説明する。続いて 4.4, 4.5, 4.6 節で、それぞれ本手法の解析手法、コード生成方法、および実装について述べる。最後に、4.7 節で性能評価の結果について考察し、4.8 節で結論を述べる。

4.2 問題点

KL1 では、動的なデータ構造生成やゴール・スケジューリングのため、データ中の通信対象部分や定義・参照を行うノードが、一般的に実行時まで判明しない。このため KLIC など現在の実装では、個々のデータが必要になった時点で送信を要求するという、要求駆動的なデータのやりとりを行っている。この結果、次のような問題点を生じている。

1. 1個のデータを受け取るという1論理メッセージのために、データ要求 (read)、返信 (answer)、および外部参照ポインタの解放 (release) と、3 個の物理メッセージが必要となる。

2. 構造型データがノード間にまたがって生成・参照される場合、どの部分が参照されるかが判らないため、個々の要素が参照される度に通信を行う必要がある。このために通信が細粒度化する。

とくに、ノード間で大規模な構造型データが転送される場合、2. が大幅な速度低下を引き起こす。

例として、図 4.1に示した、2 つのノード間でスタック操作を繰り返すプログラムを考える。ノード 1 上のゴール `stack/2` はガード部で第 1 引数を検査するため、ノード 0 にその値を要求する。しかし、図 4.2のように、参照された値がネストした構造型データであっても、ノード 1 上でどの部分が参照されるか判らないため、ネストの一番外側であるコンス・セルしか送信されない。その結果、このコンス・セルの引数を参照するために、あらためて通信が必要になる。

```
main :- drive(100,S), stack(S,none)@node(1).
drive(M,S) :- M=:=0 | S=[].
drive(M,S) :- M=\=0 | S=[push(int(M))|S0], S0=[pop(D)|S1], drive1(D,S1).
drive1(D,S1) :- D=int(N) | N1:=N-1, drive(N1,S1).
stack([]          ,D      ) :- terminate(D).
stack([push(X)|S],D       ) :- stack(S,p(X,D)).
stack([pop(X)|S] ,p(Y,D1)) :- X=Y, stack(S,D1).
terminate(D).
```

図 4.1: プログラム stack

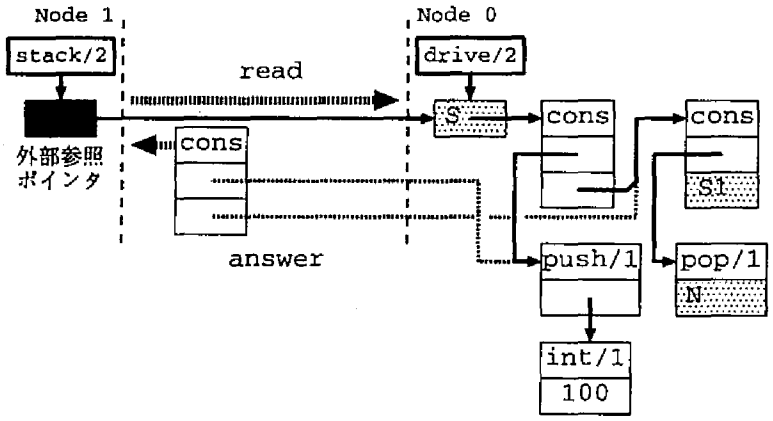


図 4.2: 構造型データの送信

この問題への対策として、現在の KLIC にはバッチ転送モードが実装されている [15]。これは、最初の要素が参照された時点で構造型データ全体を送信することにより、通信回数を減らすというものである。しかしこの方式では、送信した構造型データ内に受信側で参照しない要素があった場合、その部分を送信する時間と受信側で占めるメモリが無駄になる。図 4.2の例では、ノード 1 で参照しない int(100) まで送信してしまう[1]。この結果、大規模なデータを論理的に共有するプログラムでは、データ全体を物理的にも共有することになってしまい、メモリ不足により GC の頻発を生じたり、実行不能に陥ったりする恐れがある。

4.3 手法の概要

本節では、4.2 節で述べた問題点を改善する、選択的一括送信手法について説明する。本手法では、静的解析情報を利用することで、参照側ノードで必要なデータのみをまとめて送るコードを生成し、通信の粒度を上げる。

[1]ネスト構造の特定レベルまでをバッチ転送にすることもできるが、このレベルは実行時にユーザが与えなければならず、全データに対して同じレベルが適用されてしまう。

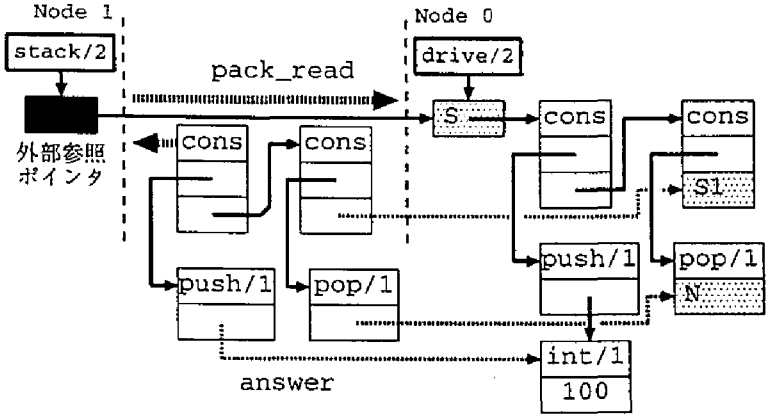


図 4.3: 構造型データの選択的一括送信

4.3.1 並列プロセス

本手法では、対象となるKL1プログラムを並列実行単位ごとに並列プロセスとして分割し、これらの並列プロセス間で選択的一括送信を行う。2.3 節で述べたように、KL1 ではプラグマ@nodeにより並列実行するゴールを指定し、それ以外の部分は同じノード上で実行される。したがって、プラグマで指定されたゴールの各々を開始点とし、そこから同一ノード上で実行される部分を、それぞれ並列プロセスとみなすことができる。

プログラムの初期ゴールである main/0、およびすべての@node プラグマ付きのゴールを並列プロセス初期ゴールとすると、並列プロセス初期ゴール g_I から他の並列プロセス初期ゴールを経由せずに呼ばれるゴールの集合が、g_I の割り当てられたノード上で逐次実行される。この集合を、並列プロセス P_{g_I} と呼び、以下 '並列プロセス main/0' のように、初期ゴール名で区別する。

4.3.2 選択的一括送信手法

2 つの並列プロセス P_p, P_c において、P_p 上の構造型データを P_c が参照する場合を考える。P_p が P_c からのデータ要求メッセージを受信したとき、構造型データ中の具体化済み部分のうち、P_c が必要とする部分のみをすべて送るようにすれば、P_c に不要なデータを送信することなくメッセージ粒度を上げることができ、通信回数が減少する。

たとえば図 4.1のプログラムの場合、コンス・セルの連鎖や、その引数であるファンクタ push/1, pop/1 は参照するが、push/1 の引数は参照しないことが判っていれば、図 4.2と同じ状況で図 4.3のようにして、1 往復の通信で具体化済み部分についての必要十分なデータを送ることができる。

本研究で提案する選択的一括送信手法では、プログラムの静的解析によりこの参照情報をコンパイル時に調べ、上記のような送信を行うための選択的一括送信コードを生成する。プログラムの実行時には、データ要求に対し、対応する送信コードを実行することにより、必要なデータの選択的一括送信を行う。

以下、静的解析および選択的一括送信コードの生成方法について、それぞれ詳しく述べる。なお、例として図 4.1のプログラム stack を用いる。

4.4 静的解析

本節では、選択的一括送信のための静的解析手法について述べる。まず、選択的一括送信の実現に必要な解析情報について考察し、続いて 3.3 節の型解析手法を基にした、本送信手法のための解析手法を説明する。

4.4.1 解析情報

まず、選択的一括送信で必要な解析情報を検討する。2 つの並列プロセス P_p, P_c において、変数 v_i が共有され、P_p がこれを構造型データ s_j に具体化し P_c がその値を参照する場合を考える。このとき、P_p から P_c への転送が必要なのは、s_j の要素のうち、P_c 内のゴールが値を参照するものである。ただし、s_j の要素の中には、P_c 内のゴールが具体化を行うものが含まれる可能性がある。たとえば、2.2.5 項で述べたようにストリーム通信では、構造型データの一種であるストリーム中のメッセージに未具体化変数を含ませ、ストリームを参照する側でこの構造型データの一部を具体化する手法が用いられる。この場合、P_c で具体化される要素は、P_c で参照も行われるものであっても、選択的一括送信の対象とする必要はない。

このように、選択的一括送信に必要なのは、各並列プロセス内で参照のみ行われるのが構造型データのどのような要素であるか、という情報である。本研究ではこれを、3章で述べた型情報を利用して、どのようなデータ型に対する参照・具体化が生じるかで表す。この型情報は名前や引数個数の異なる構造型データを別の型として区別し、引数についても再帰的に型情報を持つ。このため、4.3 節の例で述べたような参照パターンを表すことができる。

この型情報では、カウンタが一定値に達するまで入力ストリームの要素を参照する、といった、値に依存した参照パターンは表すことができない。しかしながら、多くの KL1 プログラムでは、入力ストリームが [] で閉じられるまでリストの要素を参照し続けるといった、パターンマッチングにより分岐制御を行う書き方が多用される。このため、型情報だけでも、多くの場合は充分な精度の参照パターンが得られることが期待できる。

ここで、選択的一括送信用の解析情報として、3 章の型解析に欠けている部分を考える。選択的一括送信では、ノード間の共有変数が取り得るデータ型だけではなく、そのうちのどの型が参照されるかという情報が必要である。そこで、本章では 3 章の型情報を拡張し、参照・具体化の区別を表すモードを付加した型解析を行う。また、必要なのはプログラム全体を通してではなく、個々の並列プロセス内での参照情報である。そこで、3 章のようなプログラム全体の解析ではなく、個々の並列プロセス毎に解析を行う。

以上のことより、本節では 3.3 節の型解析手法を、次のように変更・拡張する。

並列プロセス分割

本節では、まず入力プログラムを並列プロセスに分割する。プログラム中の、すべてのゴールの集合を G, g/n をクローズヘッドに持つ節の集合を $C_{g/n}$ としたとき、プログラムは図 4.4のアルゴリズムにより、並列プロセス $P_1, \ldots, P_{pmax}$ に分割される。

このアルゴリズムは、まず 2. で並列プロセス初期ゴールを見つける。また、main/0 以外は、並列プロセス初期ゴールを呼び出す疑似節:$-g(v_1, \ldots, v_n)$. を、その並列プロセスの節集合の初期値として与える。この疑似節は解析時に、親プロセスにおける初期ゴール呼出しに対応するものとして扱われる。続いて 3. で、各並列プロセスを構成する節集合を求める。初期ゴールの定義節から順に、そのボディゴールの定義節を集めていき、初期ゴールから@node プラグマを経由せず

1. $G_1 = \{\texttt{main/0}\},\ pmax = 1$
2. forall g/n@node$(m) \in G$\{
   if $G_i \neq \{g/n\}(i = 1, \ldots, pmax)$ {
   $pmax \leftarrow pmax + 1,\ G_{pmax} = \{g/n\},\ P_{pmax} = \{\texttt{:-}g(\texttt{v}_1, \ldots, \texttt{v}_n).\}$
   }
   }
3. for $i = 1$ to $pmax$ {
   while $G_i \neq \emptyset$ {
   G_iからゴール g/n を取り出す
   if $P_i \cap C_{g/n} = \emptyset$ {
   $P_i \leftarrow P_i \cup C_{g/n}$
   $C_{g/n}$のユーザ定義ボディゴールのうち@node プラグマのないものを G_iに追加
   }
   }
   }

図 4.4: 並列プロセスへの分割アルゴリズム

にたどれるゴールの定義節をすべて集め終わると、それがその並列プロセスを構成する節集合となっている。

そして、本節での型解析は、この並列プロセスそれぞれについて個別に行う。この結果、並列プロセス P_iの解析結果より P_i内で参照される型が判明すれば、その情報を用いて、他の並列プロセスから P_iへの選択的一括送信が可能になる。

たとえば、図 4.1のプログラム stack の場合、図 4.5のように、main/0 を初期ゴールとし drive/2 で再帰を繰り返すプロセスと、stack/2 を初期ゴールとし再帰を繰り返すプロセスに分割される。このそれぞれについて独立して型解析を行うことにより、並列プロセス stack/2 の参照する型情報が得られ、main/0 から stack/2 への選択的一括送信が実現できる。

モード情報の導入

本章の解析では、各型に対して具体化／参照のどちらが行われるかが、静的に区別できなければならない。そこで、3 章の型情報に対し、モード情報を付加する。

まず、本章で対象とする KL1 プログラムは、*Moded Flat GHC* [38, 39] と同様に、以下の条件を満たすものとする。

1. 各節内における同一変数の出現のうち、1 個のみが出力出現で残りはすべて入力出現である。
2. 組込述語 =/2 を除いて、述語の各引数のモードは述語名より一意に決定する。また、ファンクタなど構造データ中の引数のモードは、プログラム中におけるそのデータの出現箇所に依存するが、データ中の他の引数のモードには依存しない。

この条件を満たした上で、プログラムは well-moded [39]、すなわちすべてのモード制約を満たすモードが存在するとする。

並列プロセス main/0

```
1 main :- drive(100,S), stack(S,none)@node(1).
2 drive(M,S) :- M=:=0 | S=[].
3 drive(M,S) :- M=\=0 | S=[push(int(M))|S0], S0=[pop(D)|S1], drive1(D,S1).
4 drive1(D,S1) :- D=int(N) | N1:=N-1, drive(N1,S1).
```

並列プロセス stack/2

```
1 :- stack(S,D).
2 stack([]          ,D       ) :- terminate(D).
3 stack([push(X)|S],D       ) :- stack(S,p(X,D)).
4 stack([pop(X)|S] ,p(Y,D1)) :- X=Y,stack(S,D1).
5 terminate(D).
```

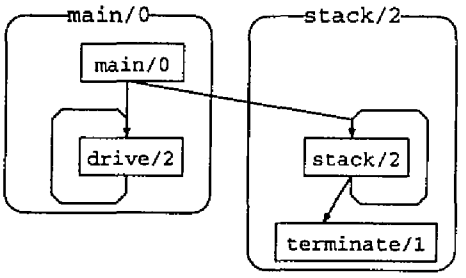


図 4.5: プログラム stack の並列プロセス分割

この well-moded なプログラムに対し、=/2 のモードと意味を考える。

通常、=/2 は引数の一方が入力、他方が出力となる。しかし Flat GHC の制約により、ガード部ではクローズヘッドの引数を具体化することができない。このため、次のような場合には、ガードゴール=/2 は両引数を入力とし、両者の同一性検査を意味する。

1. ガード部で、クローズヘッドの引数に現れる変数 v_i, v_j 間の同一化が生じる場合。または、v_i と具体値 b_l の同一化が生じる場合。

   v_i, v_j はともに具体化できないから、このゴールは同一性の検査になる。たとえば以下の節では、ガードゴール=/2 はそれぞれ X と Y、Z と 1 が同一であるかの検査となる。

   ```
   p(X,Y,f(Z)) :- X=Y, Z=1 | true.
   ```

   また、ヘッドの引数に現れない変数 $v_{k1},\ldots,v_{km}$ を介して v_i, v_j の同一化が生じる場合、一連のガードゴール=/2 のどれか一つを、同一性検査とみなす。たとえば以下の節では、3 つのゴール=/2 のうち、任意の 1 つを同一性検査、残りを通常の同一化と考えればよい。

   ```
   p(X,Y) :- X=W, Y=Z, W=Z | true.
   ```

2. ガード部で、=/2 以外の組込述語の出力引数に現れる変数 v_k と、クローズヘッドの引数に現れる変数 v_i や具体値 b_l との同一化が生じる場合。

節 C_kについて、ガードゴールの集合を G_kとすると
1. クローズヘッドの引数に出現する変数 $v_1, \ldots, v_m$について
   $V_{H,k} \leftarrow \{v_1, \ldots, v_m\}$
2. forall $g_i \in G_k$ {
     if $g_i \neq$ =/2 {
       g_iの出力引数 $v_{i1}, \ldots, v_{im_i}$について
       $V_{H,k} \leftarrow V_{H,k} \cup \{v_{i1}, \ldots, v_{im_i}\}$
     }
   }
3. $N \leftarrow V_{H,k}$
4. while $N \neq \emptyset$ {
     $v_i \leftarrow N$の要素, $N \leftarrow N \setminus \{v_i\}$
     forall $=(v_i, x) \in G_k$ {
a.   if $x \in V_{H,k} \vee x$ が具体値 {
         このゴールを=/2 → ==/2 と書き換える
       }
b.   else {
         $V_{H,k} \leftarrow V_{H,k} \cup \{x\}$, $N \leftarrow N \cup \{x\}$
       }
     }
   }

図 4.6: =/2→==/2 の書き換えアルゴリズム

この同一化ゴールで両引数は入力であるから、同一性の検査となる。たとえば以下の節では、ゴール=/2 はヘッドの引数 Y と、ガードゴールで計算した Z の値が同一であるかの検査となる。

```
p(X,Y) :- Z:=X*2, Y=Z | true.
```

このような同一化ゴールの扱いを簡単にするために、本研究では同一性検査を行う述語==/2 を導入する。==/2 は両引数を入力とし、両者が同一の値を持つなら成功、そうでなければ失敗する。値が構造型データの場合、ネストされた引数も含めて参照される。実際には、同一の未具体化変数が同じ位置に出現すればその下の部分は参照する必要がないが、本解析手法では値の全要素が参照されるものとして扱う。

以下、本研究では、対象プログラム中の上記の 1,2 に当てはまるゴール=/2 を、ゴール==/2 に書き換えてから解析を行うものとする。この書き換えは、各節に図 4.6のアルゴリズムを適用することによって行う。

また、組込述語についても、述語名から各引数のモードが一意に定まるものとする。たとえば、組込述語 arg(P,F,A) は、ファンクタ F と整数 P を入力とし、A を出力と考える。実際のプログ

ラムでは、ガードゴールで arg(1,F,0) のように、ファンクタの引数を検査するような使い方が用いられるが、これは arg(1,F,A), A==0 と書き換えることで、arg/3 が出力した A の値を==/2 で検査するとみなすことができる。また、ボディゴールで A を入力とするようなプログラムも言語仕様上は可能であるが、ボディゴールの失敗はプログラム全体の停止を引き起こすので、通常は使われることがない。したがって、上記の制限を設けても実用上は問題を生じない。

モードつき型集合

以上の仮定の下、本章では変数 v_iのモードつき型集合を、$T(v_i) = \{e_1, \ldots, e_n\}$ と定義する。各 e_jは型とモードの組 (t_j, m_j), 型変数とモードの組 (v_j, m_j), 不明要素*のいずれかである。

型および型変数については、3 章の定義と同じであるが、本章ではこれに型モード m_jを導入し、以下のように定義する。

P_k内で v_iの型 t_jの値に対し、

1. 参照のみ行われるなら $m_j = reference$。
2. 具体化のみ、もしくは具体化および参照が行われるなら $m_j = instantiation$。
3. 参照・具体化とも行われないなら $m_j = don't\_care$。

以下、本論文では型モードをそれぞれ $r, i, ?$と略記する。

並列プロセス P_k中に現れる変数 v_iについて、m_jは v_iの型 t_jである値に対する、P_k内での入出力方向を表す。v_iの出現は複数存在し得るから、t_jに対して具体化と参照が共に起こる可能性がある。しかし、他プロセスからの選択的一括送信対象となるのは、P_k内で参照のみ行われるデータ型だけであるから、そのようなデータ型を他と区別できればよい。また、並列プロセス単位の解析では、well-moded なプログラムであっても、一部の型モードが判明しない可能性がある。しかしこれは、そのような型の値に対しては P_k内で具体化も参照も行われないことを示しているから、選択的一括送信の対象外として無視できる。

また、構造型データ型の引数として現れる型変数 v_jについても、型モード m_jを考える。これは、変数 v_jのその位置への出現が、その引数の具体化・参照のどちらを意味するかを表している。

構造型データ型の引数も、3 章と同様に、再帰的に型集合で表される。こうした入れ子になった型集合についても、各要素ごとに型モードを持つため、2.2.5 項の図 2.4で挙げたような双方向通信を行うプログラムでも、正しく個々の要素の入出力方向を表すことができる。

以上の定義より、たとえばボディゴール X=f(1,Y), add(Y,1,Z) に対し、各変数のモードつき型集合は、$T(X) = \{(s(f, 2, (\{(i,i)\}, \{(Y,r)\})), i)\}$, $T(Y) = \{(i,r)\}$, $T(Z) = \{(i,i)\}$ となる。

なお、上田らのモード解析 [38, 39] でも、構造型データの引数のモードを得ることができる。このモードが変数の各出現ごとの入出力方向を表すのに対し、本手法の型モードは、ある変数の具体値となる構造型データの特定の型の要素に対して、並列プロセス全体での入出力方向をまとめて表している。選択的一括送信で必要なのは、並列プロセス全体での参照状況なので、後者の情報で必要充分である。

4.4.2 型解析の拡張

本項では、前項で定義したモードつき型集合を得るため、3 章の型解析手法を拡張・変更する。

まず、次の演算を新たに定義する。

表 4.1: 型モード間の演算

⊕演算				⊙演算			
	i	r	?		i	r	?
i	i	r	?	i	i	i	i
r	r	r	?	r	i	r	r
?	?	?	?	?	i	r	?

型モード m の融合演算　上で述べた型モードを元とする、束 $(\{?, r, i\}, \oplus, \odot)$ を定義する。元の間の順序関係は$? \subseteq r \subseteq i$であり、二項演算$\oplus, \odot$は表 4.1のように定義される。

ここで、この束の意味を説明する。無意味な値の生成が行われない場合、プログラムの実行中には、すべての具体値について具体化と参照がともに行われる。したがって、解析中の型モードの値は、「具体化・参照ともに不明」から始まり、「具体化のみ判明」または「参照のみ判明」という状態を経て、最終的に「具体化・参照ともに判明」という状態に至る。ただし、本手法ではプログラム全体の解析を行わないため、解析が終了しても最終状態に至らない型モードが存在する。このうち、「参照のみ判明」という状態に留まったものが、選択的一括送信の対象である、並列プロセス内で参照のみ行われる型となる。

また、選択的一括送信用の情報としては、「具体化のみ判明」という状態は、「具体化・参照ともに判明」という状態と区別する必要がない。したがって、この二つはまとめて型モード i としている。

この束を用いて、型集合の OR 演算と同一化を、次のように拡張する。

モードつき型集合の OR 演算　$T_{OR} = OR(T_A, T_B)$ とすると、T_{OR}は以下を満たす最小の集合である。

ただし、$m_{OR} = m_A \oplus m_B$である。

1. アトミックデータ型t_aについて

   $(T_A \ni (t_a, m_A)) \wedge (T_B \ni (t_a, m_B))$

   $\rightarrow T_{OR} \ni (t_a, m_{OR})$

   $((T_A \ni (t_a, m_A)) \wedge (T_B \not\ni (t_a, m_B)))$

   $\rightarrow T_{OR} \ni (t_a, m_A)$

   $((T_A \not\ni (t_a, m_A)) \wedge (T_B \ni (t_a, m_B)))$ についても同様

2. 構造型データ型$t_{f/n}$について

   $(T_A \ni (s(f, n, (T_{A,1}, \ldots, T_{A,n})), m_A)) \wedge (T_B \ni (s(f, n, (T_{B,1}, \ldots, T_{B,n})), m_B))$

   $\rightarrow T_{OR,k} = OR(T_{A,k}, T_{B,k})$ とおくと $T_{OR} \ni (s(f, n, (T_{OR,1}, \ldots, T_{OR,n})), m_{OR})$

   $((T_A \ni (t_{f/n}, m_A)) \wedge (T_B \not\ni (t_{f/n}, m_B)))$

   $\rightarrow T_{OR} \ni (t_{f/n}, m_A)$

$((T_A \not\ni (t_{f/n}, m_A)) \wedge (T_B \ni (t_{f/n}, m_B)))$ についても同様

3. 型変数 v_jについて

   $(T_A \ni (v_j, m_A)) \wedge (T_B \ni (v_j, m_B))$

   $\rightarrow T_{OR} \ni (v_j, m_{OR})$

   $(T_A \ni (v_j, m_A)) \wedge (T_B \not\ni (v_j, m_B))$

   $\rightarrow T_{OR} \ni (v_j, m_A)$

   $(T_A \not\ni (v_j, m_A)) \wedge (T_B \ni (v_j, m_B))$ についても同様

4. $(T_A \ni *) \vee (T_B \ni *) \rightarrow T_{OR} \ni *$

型集合間の同一化 同一化制約の結果となる型集合 T_{UNIFY}を返す関数 $unify(T_A, T_B)$ を、図 4.7 のアルゴリズムで定義する。これは、3 章の図 3.1に示したアルゴリズムに、型モードの融合処理を拡張したものである。

モードつき型解析 3 章の図 3.2に示した型解析アルゴリズムに、以下の拡張を行うことによって、モードつき型集合を求めることができる。

1. 型集合をモードつき型集合に置き換える。
2. OR 演算および関数 *unify* を、本項で拡張したものに置き換える。
3. アルゴリズム中の (b),(c) で、引数に具体値が現れるとき、それぞれ次のようにする。

   $T_{HB} \leftarrow OR(T_{HB}, \{(t(b_b), r)\})$

   $T_{BH} \leftarrow OR(T_{BH}, \{(t(b_b), i)\})$

4. $T(\mathrm{v}_i)$ が更新され、$T(\mathrm{v}_i) = \{(t_1, m_1), \ldots, (t_k, m_k), (v_{k+1}, m_{k+1}), \ldots, (v_n, m_n)\}$ となったとき、各 $(v_i, m_{i_l}) \in T(\mathrm{v}_j)$ について、$m_{i_l} = m_1 \oplus \ldots \oplus m_n$とする。

3. は、クローズヘッドやボディゴールの引数として現れる具体値に関する、型モードの追加方法である。ヘッドに具体値が現れる場合は、実際の引数とマッチするかどうかの検査になるため、型モードは rになる。一方、ボディゴールの引数に具体値が現れる場合は、新たな具体値の生成を意味するため、型モードは i になる。

また、4. は、変数 v_iに対する具体化・参照を、型変数 v_iの型モードに反映させるためである。

例として、図 4.5の並列プロセス stack/2 に、このモードつき型解析を適用する。なお、異なる節の同名変数を区別するため、X^2のように、変数名の右肩に節番号を付けて表記する。

まず、組込述語から型が判明する変数はないため、アルゴリズムの 4. が終わった段階では、すべての変数は型不明であり、Nにはクローズヘッドおよびボディゴールに直接現れる具体値だけが入っている (図 4.8(a))。次に、5. のループに入り、型情報の伝搬処理を行う。

stack/2 の第一引数は、ヘッド側がすべて具体値であることからボディ側の型集合が確定し、変数 S^3,S^4の型モードがすべて rであるため、型集合に再帰的に含まれる型変数 S^3,S^4の型モードも rに決定する。また、第二引数は、ヘッド側・ボディ側ともに具体値と変数が混ざっているため、最初のヘッド・ボディ間同一化の結果、型集合は図 4.8(b) のようになる。

1. $T_{UNIFY} \leftarrow \emptyset$ /* 返り値の初期化 */
2. **forall** $(t_i, m_i) \in T_A$ {
   if t_iがアトミックデータ型 {
   if $(t_i, m_j) \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{(t_i, m_i \odot m_j)\}$
   }
   else if $(v_k, m_k) \in T_B \vee * \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{(t_i, m_i)\}$
   }
   }
   else if $t_i = s(f, n, (T_{A,1}, \ldots, T_{A,n}))$ { /* 構造型データ型 */
   if $(s(f, n, (T_{B,1}, \ldots, T_{B,n})), m_j) \in T_B$ {
   $T_{UNIFY} \leftarrow OR(T_{UNIFY},$
   $\{(s(f, n, (unify(T_{A,1}, T_{B,1}), \ldots, unify(T_{A,n}T_{B,n}))), m_i \odot m_j)\})$
   }
   else if $(v_k, m_k) \in T_B \vee * \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{(t_i, m_i)\}$
   }
   }
   }
3. **forall** $(v_i, m_i) \in T_A$ {
   if $* \in T_B \vee (v_j, m_j) \in T_B$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{(v_i, m_i)\}$
   }
   $T(\mathrm{v}_i) \leftarrow unify(T(\mathrm{v}_i), T_B \cup \{*\})$
   }
4. T_Bについても、2,3 と同様に処理
5. **if** $(* \in T_A \wedge (* \in T_B \vee (v_j, m_j) \in T_B)) \vee (* \in T_B \wedge (* \in T_A \vee (v_j, m_j) \in T_A))$ {
   $T_{UNIFY} \leftarrow T_{UNIFY} \cup \{*\}$
   }
6. $T_{UNIFY} = \emptyset$ なら同一化失敗そうでなければ T_{UNIFY} を返す

図 4.7: モードつき型集合に対する関数 $unify(T_A, T_B)$ のアルゴリズム

$$T(\mathtt{S}^1) = T(\mathtt{S}^3) = T(\mathtt{S}^4) = T(\mathtt{X}^3) = T(\mathtt{X}^4) = T(\mathtt{Y}^4) = T(\mathtt{D}^1) = T(\mathtt{D}^2) = T(\mathtt{D}^3) = T(\mathtt{D1}^4) = T(\mathtt{D}^5) = \{*\}$$

$$N = \{[\mathtt{push}(\mathtt{X}^3)\,|\,\mathtt{S}^3],\ \mathtt{p}(\mathtt{X}^3,\mathtt{D}^3),\ [\mathtt{pop}(\mathtt{X}^4)\,|\,\mathtt{S}^4],\ \mathtt{p}(\mathtt{Y}^4,\mathtt{D1}^4)\}$$

(a) 初期値

$$T(\mathtt{X}^3) = T(\mathtt{X}^4) = T(\mathtt{Y}^4) = \{*\}$$

$$T(\mathtt{S}^1) = T(\mathtt{S}^3) = T(\mathtt{S}^4) = \{(a, r),\ (s(.,2,(\{(s(\mathtt{push},1,(\{(\mathtt{X}^3,?)\})),r),(s(\mathtt{pop},1,(\{(\mathtt{X}^4,?)\})),r)\},\{(\mathtt{S}^3,r),(\mathtt{S}^4,r)\})),r)\}$$

$$T(\mathtt{D}^2) = T(\mathtt{D}^3) = T(\mathtt{D}^5) = \{(s(\mathtt{p},2,(\{(\mathtt{X}^3,?)\},\{(\mathtt{D}^3,?)\})),i),*\}$$

$$T(\mathtt{D}^1) = T(\mathtt{D1}^4) = \{(s(\mathtt{p},2,(\{(\mathtt{Y}^4,?)\},\{(\mathtt{D1}^4,?)\})),r),*\}$$

(b) 中間結果

$$T(\mathtt{X}^3) = T(\mathtt{X}^4) = T(\mathtt{Y}^4) = \{*\}$$

$$T(\mathtt{S}^1) = T(\mathtt{S}^3) = T(\mathtt{S}^4) = \{(a, r),\ (s(.,2,(\{(s(\mathtt{push},1,(\{(\mathtt{X}^3,?)\})),r),(s(\mathtt{pop},1,(\{(\mathtt{X}^4,?)\})),r)\},\{(\mathtt{S}^3,r),(\mathtt{S}^4,r)\})),r)\}$$

$$T(\mathtt{D}^1) = T(\mathtt{D}^2) = \{(s(\mathtt{p},2,(\{(\mathtt{X}^3,?),(\mathtt{Y}^4,?)\},\{(\mathtt{D}^3,r),(\mathtt{D1}^4,r)\})),r),\ (\mathtt{D}^3,r),\ (\mathtt{D1}^4,r),\ *\}$$

$$T(\mathtt{D}^3) = \{(s(\mathtt{p},2,(\{(\mathtt{X}^3,?),(\mathtt{Y}^4,?)\},\{(\mathtt{D}^3,r),(\mathtt{D1}^4,r)\})),r),\ (\mathtt{D1}^4,r),\ *\}$$

$$T(\mathtt{D1}^4) = \{(s(\mathtt{p},2,(\{(\mathtt{X}^3,?),(\mathtt{Y}^4,?)\},\{(\mathtt{D}^3,r),(\mathtt{D1}^4,r)\})),r),\ (\mathtt{D}^3,r),\ *\}$$

(c) 最終結果

図 4.8: 並列プロセス stack/2 の型解析

その後、$D^1, D^2, D^3, D1^4$の型集合が更新されたことにより、再び第二引数のヘッド・ボディ間の同一化が繰り返され、最終的に図 4.8(c) の結果が得られる。

この最終結果を見ると、並列プロセス stack/2 中の変数が取り得る型の集合が得られていることが判る。また、並列プロセス内での入出力方向が型モードとして示され、たとえばファンクタ push/1 は参照のみ行われるが、その引数は参照も具体化もされないことが判る。

4.5 コード生成

本節では、前節での静的解析結果を用いて、実行時に選択的一括送信を行う選択的一括送信コードを生成する手法を述べる。まず、解析結果であるモードつき型集合の意味を考察し、続いて、この型集合から選択的一括送信コードを生成する方法を説明する。

4.5.1 解析結果の意味

ノード N_p上の並列プロセス P_pがノード N_c上にプロセス P_cを生成する場合を考える。また、P_cの初期ゴールを g/n とし、並列プロセス分割時に生成した疑似節を、$:\text{-}g(\mathrm{v}_1,\ldots,\mathrm{v}_n)$. とする。

P_p, P_c間の転送対象となりうるのは、両者の共有変数となる $\mathrm{v}_1,\ldots,\mathrm{v}_n$の具体値である。解析の結果、各 v_iについてモードつき型集合 $T(\mathrm{v}_i)$ が得られているから、その各要素について、次のように送信対象かどうかを判定できる。

1. $(t_j, m_j) \in T(\mathrm{v}_i)$
   $m_j = r$なら、型 t_jのデータは P_cで参照のみ行われるので、P_pからの送信対象となる。$m_j = i$ なら P_c内で具体化が行われ、m_j =?なら具体化も参照も行われないので、これらの場合は送信対象とならない。

   $m_j = r$のとき、t_jが構造型データ型 $s(f, l, (T_1,\ldots,T_l))$ ならば、各 T_kも送信対象となる可能性がある。したがって、それぞれについて再帰的に、$T(\mathrm{v}_i)$ と同様の判定を行う。

2. $(v_j, m_j) \in T(\mathrm{v}_i)$
   $m_j = r$なら、変数 v_jの具体値は P_c内で参照のみ行われるので、$T(\mathrm{v}_j)$ について $T(\mathrm{v}_i)$ と同様に判定を行う。$m_j = i, ?$のときは、1. と同様に送信対象とならないので、$T(\mathrm{v}_j)$ の判定は必要ない。

そこで、実行時に実際に生成された各 v_iの具体値について、上記の判定を行って該当部分の選択的一括送信を行うコードを生成する。

なお、逆に P_cから P_pへの選択的一括送信コードを生成するには、P_pの解析結果より P_p内のゴール $g(\mathrm{v}'_1,\ldots,\mathrm{v}'_n)$@node($N$) の引数変数の型について、上と同様に考えれば良い。

4.5.2 選択的一括送信コードの生成

本手法では、選択的一括送信コードを KL1 プログラムとして生成する。これは、選択的一括送信処理が KL1 ゴールとして実行されることから、送信データの型判定に KL1 のパターンマッチングがそのまま利用できる、必要なメモリ管理を KL1 実行系に任せることができ実装が容易になる、といった利点が得られるためである。

選択的一括送信を行う選択的一括送信ゴールは、送信対象として与えられた変数 v_iに対し、その具体値の各要素に 4.5.1 項の判定を行い、対象となる場合は送信バッファに格納する。そして、すべての要素を判定し終わった時点で、バッファの内容を送信する。

この動作を実現するために、各変数 v_i に対し、v_i を処理する節の集合 $C_{T(v_i)}$ を生成する。$C_{T(v_i)}$ 内の各節は、それぞれ次のようなものである。

1. 具体値が送信対象である場合の処理節
   4.5.1 項で送信対象と判定された型や型変数ごとに、それぞれ次の処理を行う。

   アトミックデータ型 t_j
   具体値の型が t_j なら、バッファに格納する。

   構造型データ型 $t_j = s(f, l, (T_1, \ldots, T_l))$
   具体値が f/l なら、構造型データ型名 f/l をバッファに格納し、各引数を処理する C_{T_k} を順に呼び出す。

   型変数 v_j
   v_j を処理する $C_{T(v_j)}$ を呼び出す。

2. 具体値が非送信対象である場合の処理節
   具体値の内容に関わらず、送信する必要がないため、その値への外部参照ポインタを格納する。

3. v_i が未具体化である場合の処理節
   未具体化である部分は、その時点では送信できない。したがって、必要ならば再度その部分の送信要求ができるように、v_i への外部参照ポインタを格納する。このとき、改めて v_i を要求する際にも再び選択的一括送信が行われるように、通常の外部参照ポインタの代りに、4.6 節で述べる選択的一括送信用の外部参照ポインタを格納しておく。

また、生成した節で呼び出される C_{T_k} や $C_{T(v_j)}$ が存在しない場合は、$C_{T(v_i)}$ と同様に生成を繰り返す。

たとえばプロセス stack/2 第 1 引数の場合、4.4節での解析結果 (図 4.8(c)) より、選択的一括送信コードは図 4.9のようになる。コード中の 'inline:' は、ガード部で変数の未具体化判定を行う C コードをインライン展開している。また、次節で説明するように、選択的一括送信機構はジェネリック・オブジェクトを利用して実装されており、コード中の 'generic:' は、このオブジェクトのメソッド呼出による選択的一括送信バッファの操作を表している。メソッド put_atomic, put_cons, put_functor は、それぞれの型の具体値や構造型データ型名をバッファに格納し、メソッド put_unused は送信の必要のない部分に対する外部参照ポインタをバッファに格納する。また、put_in_ref は、次節で述べる、親プロセス (main/0) から子プロセス (stack/2) への選択的一括送信を行う外部参照ポインタを、バッファに格納するメソッドである。

なお、構造型データ型の再帰構造の格納は、ボトムアップで行うようになっているが、これは、データ受信側で受け取ったデータを展開する際に、C のスタック領域があふれることを防ぐためである。

この選択的一括送信コードは、push/1, pop/1 は格納するがそれらの引数は格納しない、という 4.3.2 項で述べた動作を実現している。また、解析時にはコンス・セルの再帰構造による線形リスト構造を認識していないが、コード生成時に格納節の再帰呼び出しが生成されるため、具体化されている分はすべて一度に送られるようになっている。

実際に選択的一括送信を行うには、上記の方法で得られた選択的一括送信コードを適当なタイミングで実行するように、従来の KL1 処理系を変更する必要がある。この実装については、次節で述べる。

```
packsend_end(B)
 :- generic:sendbuf(B).
packsend_stack_2_1_top(V,BIn)
 :- packsend_stack_2_1(BIn,V,B0),
    packsend_end(B0).
packsend_stack_2_1(BIn,V,BOut)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_ref(BIn,V,BOut).
otherwise.
packsend_stack_2_1(BIn,V,BOut)
 :- V = []
  | generic:put_atomic(BIn,V,BOut).
packsend_stack_2_1(BIn,V,BOut)
 :- V = [V0|V1]
  | packsend_stack_2_1_cons(BIn,V0,B0),
    packsend_stack_2_1(B0,V1,B1),
    generic:put_cons(B1,V,BOut).
packsend_stack_2_1_cons_top(V,BIn)
 :- packsend_stack_2_1_cons(BIn,V,B0),
    packsend_end(B0).
packsend_stack_2_1_cons(BIn,V,BOut)
 :- inline:"guard_unbound(%0,%f)":[V+any]
  | generic:put_in_ref(BIn,V,BOut).
otherwise.
packsend_stack_2_1_cons(BIn,V,BOut)
 :- V=push(V0)
  | generic:put_unused(BIn,V0,B0),
    generic:put_functor(B0,V,BOut).
packsend_stack_2_1_cons(BIn,V,BOut)
 :- V=pop(V0)
  | generic:put_unused(BIn,V0,B0),
    generic:put_functor(B0,V,BOut).
```

図 4.9: stack/2 第 1 引数の選択的一括送信コード

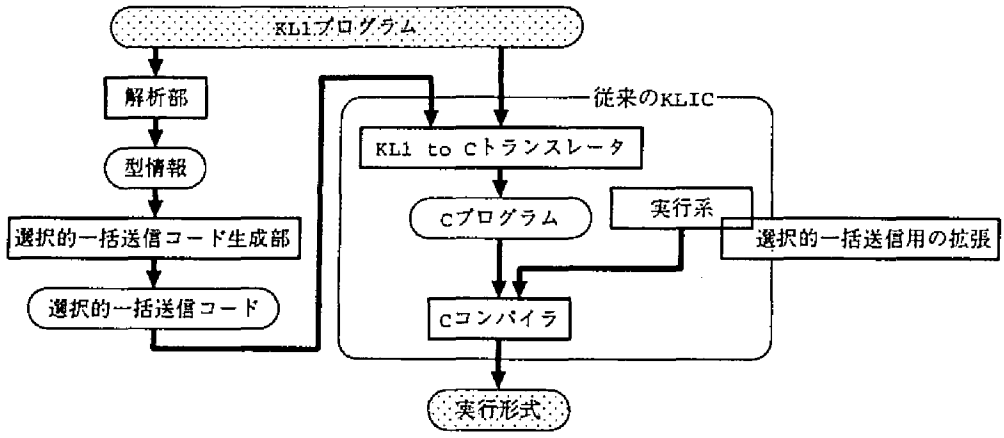


図 4.10: 選択的一括送信を行う KL1 処理系

4.6 実装

本節では、4.5 節の方法で生成した選択的一括送信コードを用いた、選択的一括送信の実装について述べる。なお、本節の内容のうち、実行系の設計・実装は伊川雅彦氏による [9, 43]。

4.6.1 拡張処理系の構成

本研究では、KLIC に対し拡張・変更を加える形で、処理系の実装を行った。処理系全体の構成を、図 4.10に示す。以下、本節では実装した処理系を拡張版処理系、元の KLIC 処理系を従来版処理系と呼ぶ。

拡張版処理系では、入力された KL1 プログラムに対し 4.4 節で述べた静的解析を行い、得られた型情報から、4.5 節で述べた選択的一括送信コードの生成を行う。この選択的一括送信コードと元プログラムを従来版のトランスレータ[2] により C プログラムに変換してコンパイルし、選択的一括送信用の拡張・変更を加えた実行系とリンクすることにより、実行形式が得られる。以下、実行系の拡張について詳しく述べる。

4.6.2 ノード間の送信モード

ノード N_p上の並列プロセス P_pが、ノード N_c上に並列プロセス P_c(初期ゴール $g(\mathrm{v}_1,\ldots,\mathrm{v}_n)$)を生成する場合を考える。このとき、$P_p, P_c$間の選択的一括送信は、次の動作で実現できる。

1. P_cで v_iへの参照が生じたとき
   2.3.4項で述べたように、N_cから N_pへデータを要求する read メッセージが送られる。このメッセージに、要求データに対応する選択的一括送信コードへのポインタを格納しておく。N_pは read メッセージを受信すると、ポインタで示された選択的一括送信コードを実行し、返信を選択的一括送信で行う。
   ここで、N_cから N_pへ選択的一括送信コードへのポインタを送るのは、N_p側が実行すべき選択的一括送信コードが、静的には決まらないためである。たとえば、P_cがさらに並列プロ

[2]後述するように、ゴール送信時に選択的一括送信コードへのポインタを合わせて送信するよう、コード生成部を若干変更してある。

セス P_gを生成し、これに変数 v_iを引数として渡した場合、P_gからも v_iの値要求メッセージが N_pに送られる可能性がある。したがって、データを要求する側の並列プロセスが、自分の必要なデータに対応する選択的一括送信コードを指定する必要がある。

2. P_cで v_iへの具体化が生じたとき
2.3.4項で述べたように、N_cから N_pへ同一化を要求する unify メッセージが送られる。したがって、P_pが必要とする v_iの値を送る選択的一括送信コードを N_c上で実行し、unify メッセージとして送信すれば良い。

well-moded を仮定しているため、v_iの具体値の送信が生じるならその方向は必ず静的に決まり、向きが決まらない場合は P_p, P_c間では送受信が生じない。また、v_iがアトミックデータ型の値しかとらない場合は、選択的一括送信を行う意味がないので、従来の送信方法で良い。以上のことより、v_iの送信モードを次のように定義する。

1. P_cの解析結果について、以下のどちらかが成り立つ場合、v_iは P_cへ選択的一括送信する必要がある。この場合の、ノード間送信ゴール g/n における v_iの送信モードを *in* とする。

   (a) $(t_j, r) \in T(v_i)$ となる構造型データ型 t_jが存在する

   (b) $(v_j, r) \in T(v_i)$ かつ $(t_k, r) \in T(v_j)$ となる構造型データ型 t_kが存在する

2. 同様に、P_pの解析結果について、上のどちらかが成り立つ場合、v_iは P_pへ選択的一括送信する必要がある。この場合の、v_iの送信モードを *out* とする。

3. 1,2 のどちらも成り立たない場合は、P_p, P_c間で構造型データの送信が行われない。この場合は、v_iの送信モードを *normal* とする。

4.6.3 実行系の拡張

前項で述べた送信モードを用いて、従来版処理系に対し、次のような拡張を行った。

1. ノード間のゴール送信時に、各引数 v_iの送信モード情報も送る。また、v_iの送信モードが *in*、*out* のとき、それぞれ P_c, P_pの解析結果より得た v_iの選択的一括送信コードへのポインタを送る。

2. N_c側では、ゴール受信時に、未具体化変数については外部参照ポインタを生成する。このとき、送信モードが *in*、*out* の変数については、それぞれ送られてきた選択的一括送信コードへのポインタを持った外部参照ポインタ `IN_EXREF`, `OUT_EXREF` を生成する。送信モードが *normal* の変数については、従来版と同じ外部参照ポインタを生成する。

3. 新たなノード間メッセージ pk_read, answer_value を設け、pk_read メッセージを受信したら、要求された値を answer_value メッセージとして選択的一括送信する。

以下、具体的な動作を説明する。

送信モード *in* の変数への参照

N_c上で v_iの値が参照されると、IN_EXREF の generate メソッドが呼び出される。このメソッドは、通常の read メッセージの代りに、IN_EXREF が保持する選択的一括送信コードへのポインタを格納した pk_read メッセージを送信する。また、N_c上では v_iの具体化は生じないことが判っているため、N_pは pk_read メッセージを受信した段階で、この外部参照ポインタが解放されたとみなす。これによって、release メッセージの送信が不要になり、4.2 節で述べたノード間通信の問題点 1 も軽減することができる。

N_pは pk_read メッセージを受信すると、v_iが具体化済であれば返信処理用のコンシューマ・オブジェクト ANSWER_HOOK を生成する。また、pk_read で指定された選択的一括送信コードより選択的一括送信ゴールを生成し、ゴールレコードキューの先頭に挿入する。v_iが未具体化であれば、コンシューマ・オブジェクト PK_REPLY_HOOK を生成して v_iにフックする。このオブジェクトは、後に v_iが具体化されて unify メソッドが呼び出されると、REPLY_HOOK のように通常の返信処理を行う代りに、上記の ANSWER_HOOK と選択的一括送信ゴールの生成を行う。

選択的一括送信ゴールは引数として v_iと ANSWER_HOOK を持ち、4.5 節で述べたように、v_iの具体値をたどりながら、選択的一括送信の対象となる要素を選択的一括送信バッファに格納していく。格納が終了すると、ANSWER_HOOK を具体化して終了する。この結果、ANSWER_HOOK の unify メソッドが呼び出され、選択的一括送信バッファの内容を answer_value メッセージとして N_cに送信する。

送信モード *out* の変数への具体化

N_p上で v_iが具体化されると、OUT_EXREF の unify メソッドが呼び出される。このメソッドは、まず選択的一括送信による値送信を行うコンシューマ・オブジェクト UNIFY_HOOK を生成する。続いて、OUT_EXREF が持つポインタで示された選択的一括送信コードより、選択的一括送信ゴールを生成してゴールレコードキューの先頭に挿入する。

この選択的一括送信ゴールは v_iと UNIFY_HOOK を引数として持ち、送信モード *in* の場合と同様に、v_iの具体値をたどりながら選択的一括送信の対象となる要素を選択的一括送信バッファに格納し、最後に UNIFY_HOOK を具体化する。この結果、呼び出された UNIFY_HOOK の unify メソッドは、選択的一括送信バッファの内容を unify メッセージとして N_pに送信する。

選択的一括送信バッファの実装

4.5.2 項で述べたように、選択的一括送信コードを KL1 のゴールとして実行することにより、型判定やメモリ管理に KL1 実行系の機能を利用できる。その反面、送信用のメッセージを格納する選択的一括送信バッファを排他制御しなければならないという問題が生じる。

従来版ではメッセージの生成・送信、および受信・解読がそれぞれ C の関数として実現されているため、あるメッセージの処理中に他のメッセージ処理が割り込むことがない。このため、ノード間メッセージを格納するバッファを、ノード毎に 1 つずつ静的に確保し、排他制御機構は設けていない。しかし、選択的一括送信では一つの選択的一括送信処理が複数の KL1 ゴールにより行われるため、ある送信メッセージ MES_iの生成中に他の送信が発生して MES_jを生成しようとすると、MES_i, MES_jを生成するゴール群が互いに競合し合う可能性がある。このため、従来版のメッセージバッファは利用できない。

単純な解決方法としては、バッファそのものをジェネリック・オブジェクトとして動的に生成

する方式が考えられる。この方式では、MES_i, MES_j は互いに独立したバッファを持つことになるため、複数の選択的一括送信処理が重なりあっても、問題を生じない。また、バッファはヒープ上にとられるから、メモリ管理はKLIC実行系に任せることができる。しかし、選択的一括送信のメッセージはある程度大きなものになるから、送信のたびにヒープ上にバッファ・オブジェクトを生成すると、ガーベジコレクションの頻度を上げることになり、実行効率が低下する。

そこで、本研究の実装では、ノードごとに1つずつ選択的一括送信用のバッファを確保し、このバッファに対する排他制御を行うことでメッセージの上書きを避ける、という方法をとっている。排他制御は、選択的一括送信ゴールキューを用意し、MES_iを作る送信ゴール g_iが生成されたとき、MES_jを作る送信ゴール g_jが実行中であれば、g_iはいったんこのキューにつながれる。g_j(およびその子孫ゴール群) は、MES_jを完成させて送信を終えると、選択的一括送信ゴールキューの先頭ゴールをゴールレコードキューの先頭に挿入する。この機構によって、一度に一種類のメッセージを作る選択的一括送信ゴールしか実行されないことが保証される。

4.7 性能評価

本節では、前節の実装による通信最適化手法の性能評価について述べる。

4.7.1 評価環境

評価用の処理系として、次の2種類を用いた。

1. PVM版KLIC+イーサネットで結合した2台のワークステーション (SS10)

2. AP1000版KLIC+64台構成のAP1000

分散メモリ型並列計算機AP1000 [44, 45] はトーラス網で結合した最大1024個のセル (プロセッサ)、およびホストとなるワークステーションから構成され、計算性能に対し通信性能が比較的高いという特徴を持つ。

2. の処理系は、ICOTより公開されている分散メモリ並列版KLIC処理系を、本研究室で移植したものである。AP1000ではフルセットのPVMが動作しないため、オリジナルKLICのPVM関数による通信部分を、同等のAP1000 Cライブラリ関数 [46, 47] に置き換えた。また、割込機能に制限があるため、プロセッサ間メッセージの到着は、1リダクションごとにポーリングを行う形に変更した。

AP1000では、buffer receiving [48] を用いることで、到着メッセージは自動的にリングバッファに格納される。このリングバッファの書込位置ポインタを監視することで、比較的効率の良いポーリングが可能である [49]。

なお、ワークステーション、AP1000のいずれの場合も、各プログラムごとに示す台数のプロセッサの他に、KLICの並列実行管理機能に1プロセッサを割り当てている。

4.7.2 評価プログラム

評価プログラムには、次の5種類 (ベンチマーク3本、応用2本) を用いた。

stack 図4.1に示した、2ノード間でスタック操作を行うプログラムである。繰り返し回数は1. で1000、2. で10000とした。

mastermind 数当てゲームで、一定回数以内の試行で正解に至るような推測の並びを、全解探索する。探索の途中で分岐が生じるたびに隣のノードから順にゴールを割り当てるため、細粒度の負荷分散が行われる。

nqueen N-queen 問題は、チェスの駒の一つクイーンを、お互いに相手を取ることの出来ない位置を占めるよう、$N \times N$の盤面上にN個配置するという問題である。

本プログラムは、この条件を満たす配置を全解探索するものであり、負荷分散は最初の分岐のみで行われる。このため、粗粒度の負荷分散となる。

また、問題のサイズは$N = 12$としている。

pia 反復改善法を用いたマルチプル・シーケンス・アライメント・プログラム [12] であり、一回の改善ごとに、アライメント処理を並列実行する。問題のサイズは長さ 30 の配列 17 本とした。

cmgtp モデル生成型の定理証明システムで準群問題 (QG5 のオーダ 8) を解くプログラム [13, 50] である。負荷分散はマスタースレーブ方式を用いており、暇なノードに仕事が割り当てられる。

4.7.3 実行結果

各プログラムについて、通常の送信、4.2 節で述べたバッチ転送モード、選択的一括送信、の 3 通りについて、それぞれ実行時間と処理系全体の物理通信回数、および総転送量を測定した。結果を表 4.3, 表 4.2 に示す。ただし、通信オーバヘッドが非常に大きい mastermind と、マスタースレーブ方式のため並列実行に 3 台以上が必要な cmgtp については、AP1000 上の結果のみを挙げてある。

本手法を用いたことにより、pia 以外では通信回数が数分の 1 から数十分の 1 に減少している。この結果、各メッセージごとに必要なヘッダ情報などが削減されるため、通信量も通常の送信より大幅に減少している。

stack や mastermind のように通信回数が多いプログラムの場合は、2〜4 倍程度の速度向上が得られた。これらはベンチマーク用の極端なプログラムであるため、本手法を用いても逐次に対する速度向上は得られない。しかし、ある程度並列性の期待できる大規模プログラムでも局所的に通信ネックを生じる可能性があるから、本手法はそのような場合の改善に役立つと考えられる。

一方、相対的に通信回数の少ない他の 3 本も、多くの場合は速度向上が得られた。とくに、AP1000 上の cmgtp、SS10 上の nqueen では、それぞれ約 3 倍、約 5 倍の速度向上が得られ、通常の通信では得られなかった台数効果を引き出すことができた。pia については AP1000 上でかえって速度が低下したが、これは選択的一括送信では処理データが一度に届く代わりに、それまで若干の待ち時間ができるのに対し、通常の送信では送信完了までの時間は長くても、届いた分から処理を開始できるため、と考えられる。実際に、アラインメント処理の対象となる文字列のリストを従来の送信方式に戻し、これ以外のパラメータを選択的一括送信するように変更すると、実行時間は 21.4 秒まで改善できた。一方、通信速度の遅い SS10 上では、この待ち時間よりも通信回数の減少の効果が大きいため、2 倍以上の速度向上が得られている。

バッチ転送との比較でも、多くの場合は選択的一括送信の方がよい性能を出している。mastermind や cmgtp のようにバッチ転送の方が通信回数を削減できているものもあるが、参照されないデータまで送信してしまうため、通信量は選択的一括送信より多くなっている。mastermind などで

表 4.2: SS10+イーサネットでの実行結果

プログラム		逐次	通常	バッチ	一括
stack	時間 (秒)	0.06	96.3	31.4	22.7
(2 ノード)	通信 (千回)	—	13.0	4.44	3.01
	転送 (Kbyte)	—	367	167	152
nqueen	時間 (秒)	68.8	194	36.5	37.3
(2 ノード)	通信 (千回)	—	22.3	0.052	0.79
	転送 (Kbyte)	—	653	1084	152
pia	時間 (秒)	109	404	174	163
(2 ノード)	通信 (千回)	—	59.1	20.8	18.7
	転送 (Kbyte)	—	1889	1209	1016

表 4.3: AP1000 での実行結果

プログラム		逐次	通常	バッチ	一括
stack	時間 (秒)	1.56	51.6	30.1	21.1
(2 ノード)	通信 (千回)	—	136	70.0	30.0
	転送 (Kbyte)	—	3741	2266	1524
mastermind	時間 (秒)	3.44	22.9	139	7.59
(32 ノード)	通信 (千回)	—	216	9.90	43.5
	転送 (Kbyte)	—	6697	48580	2885
nqueen	時間 (秒)	412	52.4	41.2	36.2
(12 ノード)	通信 (千回)	—	43.4	0.27	0.21
	転送 (Kbyte)	—	1265	2096	276
pia	時間 (秒)	248	23.7	25.7	28.2
(18 ノード)	通信 (千回)	—	65.7	55.1	44.5
	転送 (Kbyte)	—	2087	1891	1807
cmgtp	時間 (秒)	43.4	42.4	15.2	14.2
(8 ノード)	通信 (千回)	—	28.9	0.22	0.61
	転送 (Kbyte)	—	953	411	376

は通常の通信よりも大幅に通信量が増えた結果、大きな速度低下を生じており、このような事態を避けるためには、本手法は有効であると言える。

4.8 結言

本章では、並列論理型言語 KL1 のメッセージ通信最適化手法として選択的一括送信を提案し、その内容と実装、性能評価について述べた。

本手法では静的解析情報を用いて、通信対象となるデータ型や通信方向をコンパイル時に決定する。そしてこの情報を利用して、受信側で参照されるデータのみを選択的一括送信するような通信コードを生成することにより、余分なデータ転送を生じずに粒度の高い通信を実現する。

ベンチマークや応用プログラムによる性能評価の結果、大幅な通信回数の削減を達成できた。また、多くの場合において数倍の速度向上が得られた。とくに、イーサネットのように通信オーバヘッドが大きい環境や、大規模なデータ転送が行われるプログラムでは、本手法により、従来の通信方法では得られなかった並列効果を引き出すことができ、このような場合に対する最適化手法としての有効性が確認できた。

しかしながら今回の性能評価でも、pia のように選択的一括送信によるデータ到着までの待ち時間が若干の性能低下につながる場合があった。これを解決するには、選択的一括送信を行うデータサイズに上限を設け、大規模データの場合は何度かに分割して送るなどの工夫が必要である。また、細粒度のまま通信オーバヘッドを削減する単方向送信 [51] の適用も有効と考えられる。

また、選択的一括送信ではデータ要求を受けた段階で具体化されている部分だけを対象としているため、参照が具体化より先に届くことが多いプログラムでは効果が期待できない。この問題は、具体化がある程度進むまでデータ送信を遅らせる遅延送信により解決できると考えられる。

こうした選択的一括送信の改良については、現在までに黒田により、転送粒度の動的制御を行う手法が提案されている [52]。これは実行時に、一回のデータ転送量とノード間データ要求の応答時間を記録し、転送量を動的に変化させるというものであり、性能評価により、選択的一括送信の性能を改善できることが確認されている。

Chapter 5

ゴール・スケジューリングの最適化

5.1 緒言

前章では、分散メモリ型並列計算機上の KL1 処理系で、メッセージ通信を最適化する手法を述べた。しかし、KLIC など現在の KL1 処理系では、単一ノード上で実行される部分にも余分なオーバヘッドが多く、逐次実行性能も C など他の言語の処理系に及ばない。このオーバヘッドを削減することは、ワークステーションなど逐次環境下での速度改善だけではなく、個々のノード上での実行速度を上げることにより並列実行の性能向上にもつながる。

そこで、本章では、KL1 処理系の逐次実行部におけるオーバヘッドの主要な要因である、ゴール・スケジューリングを最適化する手法を述べる。前章と同様に、静的解析情報を利用することによって、従来は動的に行っていたゴール・スケジューリングをある程度までコンパイル時に行い、動的オーバヘッドを削減することができる。また、動的スケジューリングにも解析情報を利用することで、さらに効率的な実行が可能になる。

以下、5.2 節で従来の実装における問題点を述べ、5.3 節でこれを解決する手法の概要を説明する。続いて 5.4, 5.5 節で、それぞれ本手法の解析手法および実装について述べる。その後 5.6 節で性能評価の結果について考察し、5.7 節で関連研究との比較を行い、最後に、5.8 節で結論を述べる。

5.2 問題点

5.2.1 従来のゴール・スケジューリング方式の問題点

2.3.2 項で述べたように、KL1 では未具体化変数への参照が起きると実行ゴールが中断し、ゴール・スケジューラは他の実行可能ゴールに制御を移す。この未具体化変数が他のゴールにより具体化されると、中断ゴールは実行を再開する。こうした中断・再開機構により、ゴールが正しい順序で実行されることが保証されるが、この処理が頻繁に行われると、そのオーバヘッドが無視できなくなる。また、ゴール環境 (KLIC ではゴールレコード) をヒープ上に確保する処理や、生成されたゴールを実行可能ゴールキューにつないだり、次の実行対象としてキューから取り出したりする処理も、オーバヘッドとなる。

従来の KL1 処理系でも、これらのオーバヘッドを削減する工夫はなされている。しかし、静的解析情報は利用していないため、出現頻度が高いと思われるプログラムパターンを想定し、このパターンに適したゴール・スケジューリング戦略を適用している。このため、処理系の採用して

```
1 main :- p[1](10000,L), q[2](L).
2 p(N,L) :- N>[3]0 | L=[4][mes(N,NN)|NL], NO:=[5]NN-3, N1:=[6]NO+2, p[7](N1,NL).
3 p(N,L) :- N=<[8]0 | L=[9][].
4 q([mes(N,NN)|L]) :- NN=[10]N, q[11](L).
5 q([]).
```

図 5.1: プログラム handshake1

いる戦略に対象プログラムが適合する場合は比較的効率の良い実行が行われるが、適合しない場合は大幅な速度低下を生じる。

以下、従来の処理系で実装されてきたスケジューリング方式と、それらの問題点を考察する。

プロセス指向スケジューリング

Multi-PSI [16] や PIM [18] 上の KL1 処理系が採用している方式であり、実行中のゴールの子ゴールを優先的にスケジューリングすることで、ゴールの切替オーバヘッドを減らす戦略をとっている。

このスケジューリング方式では、ゴールの中断を生じない限り、ゴールの呼出し木を左深さ優先順にたどる形で実行が行われる。したがって、データフローがこの流れに沿っているプログラムについては、効率的に実行できる。しかし、以下のような場合には、大きなオーバヘッドを生じ得る。

1. 一度、ある並行プロセスのゴールが選択されると、その並行プロセス内に実行可能ゴールが存在する限り、それらのゴールが実行される。このため、並行プロセス間の切替は必ずゴールの中断を伴う。したがって、細粒度の並行プロセスが頻繁に切り替わりつつ交互に実行されるようなプログラムでは、切替時のゴール中断オーバヘッドが無視できないほど大きくなる。

2. ゴール g_iが中断すると、実行系のゴール・スケジューラは、深さ優先順に他の実行可能ゴールをスケジューリングしようとする。この時にゴール間のデータ依存は考慮されないため、これらのゴールに g_iの出力を参照するものがあっても、いったんスケジューリングしてから実行不能であることが判り、中断して他のゴールをスケジューリングしなおす、という動作を行う。したがって、g_iにデータ依存するゴールが多数ある場合、こうした無駄なスケジューリングと中断のオーバヘッドが大きなものになる。

たとえば、図 5.1のプログラムでは、2 個の並行プロセス p/2 と q/1 が、1 本のストリームにより交互に整数値をやり取りする。このプログラムをプロセス指向方式で実行すると、次の順にゴールが実行される。

$$1 \rightarrow 3 \rightarrow 4 \rightarrow 5(\text{中断}) \rightarrow 6(\text{中断}) \rightarrow 7(\text{中断})$$

ゴール 4 でプロセス q/1 へのメッセージを具体化するが、メッセージの引数の一つである NN は q/1 側で具体化されるため、NN の値を参照する次のゴール 5 は中断する。続いて、ゴール 6, 7 が順にスケジューリングされるが、それぞれゴール 5 の出力である NO、ゴール 6 の出力である N1

に依存するため、いずれも中断する。そして、プロセス p/2 側に実行可能ゴールがなくなったため、プロセス q/1 側に制御が移り、次の順にゴールが実行される。

$$2 \rightarrow 10(5 \text{ が実行再開}) \rightarrow 11(\text{中断})$$

ゴール 11 がガード部でストリームの次の要素を参照しようとして中断すると、次に実行可能なゴールは、ゴール 10 によって実行可能になったゴール 5 しかない。このため、制御は再びプロセス p/2 側に移り、次の順に実行される。

$$5(6 \text{ が実行再開}) \rightarrow 6(7 \text{ が実行再開}) \rightarrow 7$$

以後は同様の順にゴールの実行が繰り返される。

このように、実行の過程を追跡してみると、このプログラムは、前記のオーバヘッドを生じる例になっていることが判る。まず、並行プロセス p/2 と q/1 の実行が頻繁に切り替わるが、これは必ずゴールの中断を伴っている。また、並行プロセス p/2 側では、ゴール 5 が中断した後、データ依存関係を考えれば実行できるはずのないゴール 6, 7 も次々にスケジューリングされては中断し、ゴール 5 の実行再開後に、ゴール 6, 7 も実行再開処理が行われる。こうした無駄な処理が p/2 の 1 ループごとに繰り返されるため、非常に大きなオーバヘッドを生じている。

resumption first スケジューリング

KLIC で採用されているスケジューリング方式であり、細粒度並行プロセスに対し、プロセス指向方式で生じるオーバヘッドを軽減することができる。

この方式では、基本的にプロセス指向方式と同様、左深さ優先順にゴールをスケジューリングする。ただし、変数の具体化により再開された中断ゴールは、実行中のゴールの次にスケジューリングされる。つまり、並行プロセス P_i 中のゴール g_k が変数 v_m を具体化したとき、並行プロセス P_j のゴール g_l が v_m への参照により中断しているならば、g_k の次に g_l がスケジューリングされる。この結果、たとえ P_i にまだ実行可能ゴールが残っていたとしても、並行プロセスの実行は P_i から P_j に移ることになり、P_i 中のゴールの中断を待つことなく、並行プロセスの切替が実現される。

たとえば、図 5.1のプログラムでは、並行プロセス p/2 が中断するまではプロセス指向方式と同じ動作をするが、q/1 に切り替わった後は、次の順にゴールが実行される。

$$2 \rightarrow 10(5 \text{ が実行再開}) \rightarrow 5$$

プロセス指向方式と異なり、ゴール 11 が中断する前に再び p/2 へと切り替られるため、ゴール 11 は実行可能のまま実行可能ゴールキューに留まっている。このため、並列プロセス切替時のゴール中断を半分に減らすことができる。

しかしながら、この resumption first 方式でもプロセス指向方式と同様、データ依存により実行できないゴールもスケジューリングしてしまうという問題点は解決できていない。

また、この方式では別の問題が生じ得る。ゴール g_j の中断原因となった変数 v_i が具体化されると、resumption first 方式では g_j が直後にスケジューリングされ、並行プロセス間の切替が生じる。しかし、v_i の具体値が構造型データの場合、その引数を具体化するゴールがまだ実行されておらず、未具体化である可能性がある。そのような場合、切り替わった並行プロセス内で未具体化引数が参照されると、再びゴールの中断が生じ、未具体化引数を具体化するためにもとの並行プロセスに制御を戻す必要がある。この結果、本来なら 1 回ずつのプロセス切替で済むはずが、

```
1 main :- p0^1([int(100)|L1],L2), p1^2(L1,L2).
2 p0(L1,L2) :- L1=^3[int(N)|NL1]
             | L2=^4[int(NN)|NL2], NN:=^5N-1, p0^6(NL1,NL2).
3 p0(L1,L2) :- L1=^7[] | L2=^8[].
4 p1(L1,L2) :- L2=^9[int(N)|NL2], N>^10 0
             | L1=^11[int(NN)|NL1], NN:=^12N-1, p1^13(NL1,NL2).
5 p1(L1,L2) :- L2=^14[int(N)|NL2], N=:=^15 0 | L1=^16[].
```

図 5.2: プログラム handshake2

構造型データが完成する前に参照側に制御を戻してしまうため、何度も切替を繰り返す、という無駄な動作をしてしまう。

たとえば、図 5.2のような、2 個の並列プロセス p0/2, p1/2 が 2 本のストリームで整数値をやり取りするプログラムを考える。

このプログラムを resumption first 方式で実行すると、まず次の順にゴールが実行される。

$$1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6(\text{中断}) \rightarrow 2 \rightarrow 9 \rightarrow 10 \rightarrow 11(6\text{が実行再開})$$

はじめに並行プロセス p0/2 が実行され、ゴール 6 の中断によりプロセス p1/2 に切り替わる。そして、ゴール 11 の実行により中断ゴール 6 が実行再開すると、resumption first であるため、p1/2 側の実行可能ゴールであるゴール 12, 13 を後回しにして、ゴール 6 がスケジューリングされ、p0/2 側に制御が移る[1]。

$$6 \rightarrow 3 \rightarrow 4 \rightarrow 5(\text{中断}) \rightarrow 12(5\text{再開}) \rightarrow 5 \rightarrow 6$$

p0/2 側では、ゴール 3, 4 と実行が進むが、次のゴール 5 で N の値を参照した時点で、これが未具体化であるため中断してしまう。その後、次の実行可能ゴールであるゴール 12 がスケジューリングされ、この実行により N が具体化されると、ゴール 5 が実行再開し、resumption first により即座に次の実行ゴールとなる。

このように、resumption first 方式でこのプログラムを実行すると、ストリームメッセージを 1 往復させるごとに 4 回のプロセス切替を行ってしまい、無駄なオーバヘッドを生じてしまう。

5.2.2 ゴールのスレッド化における問題点

KL1 のような細粒度言語の実装で動的スケジューリングのオーバヘッドを減らすには、複数のスケジューリング単位を 1 スレッドに融合する手法が効果的である。たとえば、関数型言語 Id90 では、この手法による高速化が図られている [53]。

この手法は基本的に KL1 にも応用できるが、以下に挙げるような論理型言語の特性により、いくつかの問題が生じる。

1. 論理型言語では、論理的なデータフローが必ずしもゴール間の依存を表してはいない。

2. 構造型データの場合、双方向のデータフローを持つ可能性がある。

[1]現在の KLIC の実装では、ゴールのリダクション時にボディゴールのうち実行可能な組込述語をその場で実行するため、この例ではゴール 11, 12 はまとめて実行され、ここでいう問題は生じない。しかし、これらのゴールが組込述語ではなくユーザ定義述語の形で書かれていれば、上記の冗長なプロセス切替が生じる。

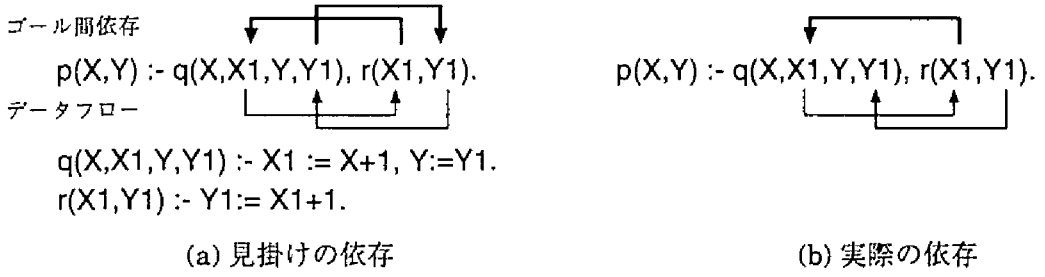


図 5.3: データフローとゴール間依存

ゴール間の依存

KL1 ではゴールの引数の入出力方向は明示的に与えられないが、3.2節で述べたプログラムのモード解析 [38, 39] を行うことで得ることができる。そして、モードにより示されたデータフロー順にゴールを実行することにより、ゴールの中断回数を減らすことができる。

しかし、モード情報が表す論理的なデータフローは必ずしもゴール間の依存を表してはおらず、ゴールの実行順序制約としては不必要に厳しい可能性がある。たとえば、整数加算を行う組込述語 add/3 は、入力引数が具体化されるまで実行できないから、データフローがそのまま依存を表している。一方、二つの変数間の同一化ゴール v_i=v_jは、意味的には変数 v_iから v_j、もしくはその逆方向のデータフローを表す。しかし、KL1 の実行モデルの上では、同一化の際には v_iや v_jの値を必要とせず、v_i, v_jが未具体化のままでもこのゴールは実行できる。つまり、この場合はデータフローがそのまま依存にはなっていない。

図 5.3の例では、モード情報からは、(a) のような双方向のゴール間依存が存在するように見える。しかし実際には、データフロー上 r/2 から q/4 に流れる Y1 の値は、q/4 の子ゴールでは参照されない。したがって、実際の依存は (b) のようになる。

構造型データのデータフロー

論理型言語では、構造型データの引数として未具体化変数をとることができ、そうした変数を後から個別に具体化することができる。この性質により、2.2.5 項で述べたように、一本のストリームによる双方向通信のような柔軟なプログラミング手法が可能になっている。しかしその反面、単純にゴールの引数のモード情報だけでは正確なデータフローを表現できず、ひいてはゴール間の正確な依存情報を得ることも難しくなっている。たとえば4.2 節で示したプログラム stack は、ゴール stack/2 の引数がともに入力であるため、見掛け上は stack/2 から drive/2 への依存しかないように見える。しかし、実際は第一引数であるストリームの要素 pop/1 の引数について、drive/2 から stack/2 への依存が存在する。

この問題を解決するには、ゴールの引数のモード情報だけではなく、引数の取り得る型情報も解析し、構造型データのそれぞれの要素のデータフローや、それに伴うゴール間の依存を解析する必要がある。

5.3 手法の概要

本研究では、静的依存解析によりゴールをスレッド化することで、部分的に静的スケジューリングを行い、動的スケジューリングのオーバヘッドを削減する。相互に依存し合う並行プロセス

のように、プログラムが持つ本質的な並行性は、複数のスレッドを動的にスケジューリングすることで保たれる。また、前節で述べたゴールのスレッド化に関する問題点については、後述する静的解析手法を用いることで解決できる。

しかしながら、並行制御の粒度を大きくすることで、並行性の低下が問題となる可能性がある。とくに並列実行を行う際には、並行性の低下により、ノードによっては仕事のないアイドル時間が発生して並列性が下がり、実行速度の低下を招く可能性もある。そこで、スレッドのスケジューリング方式についても、こうした問題を解決する手法を提案する。

5.3.1 ゴールのスレッド化

まず、本研究における依存とスレッドを定義する。

依存の定義

KL1では、ゴールは組込述語またはユーザ定義述語の呼出を意味し、ゴールの実行モデルは次のように捉えることができる。

組込述語 実行系が引数の一部を参照し、引数の一部を具体化し、終了する。

ユーザ定義述語 ガード部の組込述語で、クローズヘッドの引数の一部を参照し、ヘッドに出現しない変数の一部を具体化する。その後、ボディ部を子ゴールとして生成し、終了する。

このモデルを用いて考えると、節 c_i 内のあるゴール g_j を中断なしで実行するには、次の条件を満たしている必要がある。

制御依存 ゴール g_j が生成されるためには、その親ゴール g_p が実行されていなければならない。また、g_j がボディゴールの場合は、c_i のガードゴール $g_{g1},\ldots,g_{gm}$ も実行されていなければならない。したがって、g_j はこれらのゴールに依存する。本論文では、この種の依存を、制御依存と呼ぶ。

データ依存 g_j が変数 v_k の値を参照するなら、その値を生成するゴール g_w、および、生成された値と v_k を同一化するゴール群 $g_{u1},\ldots,g_{ul}$ が、g_j に先だって実行されている必要がある。したがって、g_j はこれらのゴールに依存する。本論文では、この種の依存を、データ依存と呼ぶ。

図 5.4に、こうしたゴール間依存の例を挙げる。`add(Y,1,Z)` は、親ゴール `p(W,V)`、および、ガードゴール `X=Y, integer(Y)` が実行されるまで生成されないので、これらに対し制御依存する。また、参照する変数 `Y` の値を生成する `W=1`、および、`W` から `Y` へデータが流れる経路を構成する `X=Y` に対して、データ依存する。

これらの依存は、合わせてゴール g_j の直接依存を表している。もし、g_j の依存ゴール g_w がゴール g_q に依存するなら、g_q も g_j より前に実行される必要がある。したがって、g_j は間接的に g_q に依存するといえる。本論文では、この種の依存を間接依存と呼ぶ。

以上の定義を用いると、ゴール g_j が中断なしで実行できる条件は、「g_j が直接または間接依存するゴールすべてが実行済であること」となる。

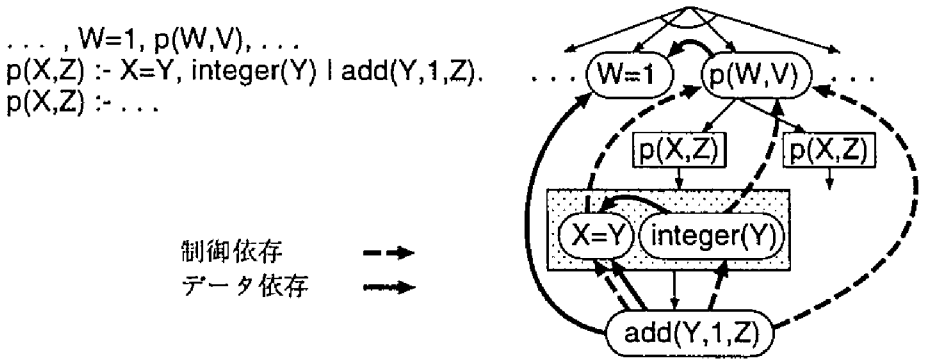


図 5.4: ゴール間依存

スレッドの定義

静的解析でゴール間の依存を得ることにより、依存関係の循環を生じないゴールの集合については、デッドロックを生じることなく静的に実行順序を決定することができる。本論文では、このようなゴールの並びをスレッドと定義する。また、スレッドの先頭ゴール p/n をスレッド開始ゴール、このスレッドをスレッド p/n と呼ぶ。

スレッド内のゴールについては、同一スレッド内の後続ゴールに対するデータ依存がないことが保証される。このため、あるスレッド内のゴールが中断したときは、後続ゴールの実行を試みることなくスレッドごと中断し、他のスレッドに実行を切り替えることができる。中断原因となった変数が具体化されると、フックしていたゴールとともに、中断していたスレッドも実行再開する。そして、このスレッドが再びスケジューリングされると、中断していたゴールから後の部分が実行される。また、スレッドの管理に実行待ちスレッドキューを用い、中断や切替、実行再開を、従来のゴールと同様に動的に行うことで、元のプログラムが持っている本質的な並行性を保つことができる。

このように、本手法のスレッドは、Id90 の研究で用いられている定義のように、スレッド全体が中断なしで実行できることを保証するものではない。そのためスレッド外への依存データに関する具体化検査などのオーバヘッドは削減できないが、スレッドの粒度を大きくすることができ、本質的に並行な部分以外の動的スケジューリングオーバヘッドをなくすことができる。

また、荒木らの研究 [54] ではゴールそのものを融合またはインライン展開してしまうのに対し、本方式ではゴールの実行順序が固定されるだけであり、実行モデルは従来と同様にゴール単位のものとなる。この方式は、プログラムを変形してゴールの数を減らす方式に比べると、ゴール環境を生成するオーバヘッドが残るという欠点がある。しかし、ゴール融合が不可能なループに対しても、逐次実行できるならば全体で 1 本のスレッドにできるという利点がある。このため、かなり粒度の大きなスレッドを生成することができ、そのぶん動的スケジューリングのオーバヘッドを小さくすることができる。この場合、逆に粒度を上げすぎることが並行・並列性の低下につながるという問題が生じるが、本方式では元プログラムのゴールの形が残っているため、スレッド内の任意のゴールを実行時に新たなスレッドとして分割することによって、動的なスレッドの粒度調整が可能である。この点については、5.3.2 項で述べる。

本手法では、相互依存性のないすべてのボディゴール間の実行順序を決定するのではなく、各節単位で、節内のボディゴール間の実行順序だけを決定する。この結果、たとえばある節のボディ

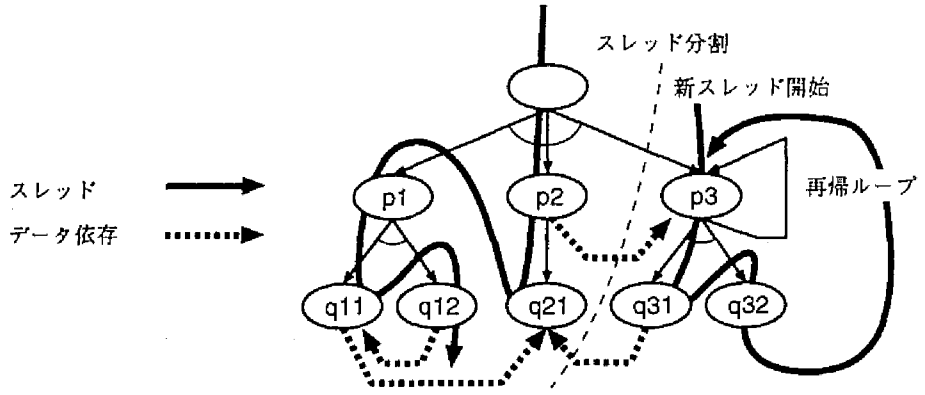


図 5.5: ゴールのスレッド化

ゴール $p_1,\ldots,p_n$のうちの p_iが実行されたとき、p_iをクローズヘッドとする節のボディゴールを $q_1,\ldots,q_m$とすると、まだ実行されていない $p_j(j \neq i)$ やその子孫ゴールは、$q_1,\ldots,q_m$やその子孫ゴールの実行がすべて終わるまでスケジューリングされない。このため、兄弟ゴール p_i,p_k間に直接相互依存がなくとも、それらの子孫間の依存も含めると、静的に実行順序を決められないことがある。この場合は p_kを新スレッドの開始ゴールとして、そこから同様に p_kの子孫ゴールの実行順を決定していく (図 5.5)。

この方式では、実際には順序決定可能なゴール群でも、複数のスレッドに分割してしまうことがある。たとえば、図 5.6の例では、従兄弟関係にあるゴールを含めれば、次のように実行順序を決定できる。

$$\mathtt{p} \rightarrow \mathtt{q1} \rightarrow \mathtt{p'} \rightarrow \mathtt{q'1} \rightarrow \mathtt{q2}$$

しかし、本手法の定義では、p の子孫と p' の子孫を交互に順序付けすることはできないので、この例では 2 本のスレッドに分割されてしまう。もっとも、こうした見掛け上の並行性を解消するようにプログラムを書き直すことは容易であり、実際にこうした記述がされることも少ないと考えられる。逆に、このように実行順序決定の範囲に制限を設けることにより、解析が容易になり、スタックを用いた効率的な実装が可能になる、といった利点が生まれる。

本手法を用いることで、同一スレッド内のゴールを順に実行する過程では、動的スケジューリングのオーバヘッドを削減できる。さらに、スレッド内のゴールについては、実行順序が固定されたことにより、従来の処理系のようにゴール環境をヒープ上に生成する代りに、WAM [55] のようにスタック上に保持することができる。ゴール環境は生成された後、実行が完了した時点で不要になるから、長期にわたる中断や実行待ちが起きない限り、一般に短命である。したがって、ゴール環境をスタック上で管理することにより、短命なデータが大量に生成・破棄されることによるヒープの消費を抑え、ガーベジコレクションの回数を大幅に減らすことができる。その他、実行可能ゴールキューが不要になるため、これの操作オーバヘッドがなくなるという利点もある。

また、5.2.1 項で述べた従来のゴール・スケジューリング方式の問題点も、本方式により解決できる。

まず、プロセス指向方式では、中断したゴール g_jに依存するゴール群が次々にスケジューリングされては中断する、という問題点があった。これを解決するには、g_jに依存するゴールをスケジューリング対象にしないようにする必要がある。本方式では、g_jに依存するゴール群は g_jと同

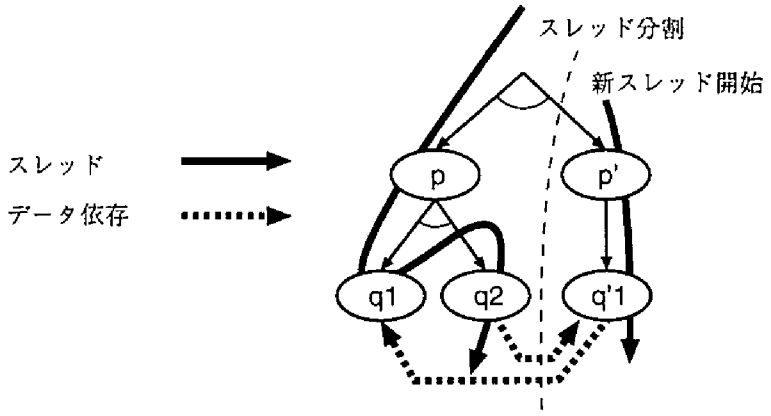


図 5.6: 見掛け上の並行性

じスレッド内に、後続ゴールとして配置される。したがって、g_jが中断した時点でこのスレッドごと中断され、依存ゴール群のスケジューリングは発生しない。一方、並行プロセスの切替が必ずゴールの中断を伴うという問題は、ゴールのスレッド化では解決しない。しかし、スレッドのスケジューリングに resumption first と同様の方式を用いることで、resumption first と同程度までの改善が可能である。これについては、5.3.2 項で詳しく述べる。

次に、resumption first 方式では、構造型データが完全に具体化される前に、そのデータを参照するプロセスに制御が移行してしまい、残りの部分を具体化するために冗長なプロセス切替が発生する、という問題があった。これを解決するには、ゴールの中断原因となった変数が具体化されても、再開された中断ゴールを直ちにスケジューリングするのではなく、デッドロックを生じない範囲で、中断プロセスへの制御移行を遅らせる必要がある。本方式では、具体化側スレッドが中断するまで参照側スレッドに制御が移行しない。このため、残りの部分を具体化するゴールが他のスレッドのゴールに依存していない限り、それらのゴールの実行前に参照側スレッドが再開されることはない。したがって、多くの場合はこの問題が解決される。

5.3.2 スレッドのスケジューリング方式

スレッド内のゴールは完全に逐次化されているため、並行プロセスのように本質的に並行性を持つ部分は、複数のスレッドを動的にスケジューリングすることで保たれる。したがって、実行効率を向上させるには、このスレッドのスケジューリングにも適切な方式を用いる必要がある。ゴール・スケジューリングと比較すると、スレッド・スケジューリングには次のような違いがある。

1. スケジューリング単位がゴールからスレッドへと粒度の大きいものになったため、スケジューリングを行う回数は相対的に少なくなっている。

2. 同じくスケジューリングの粒度が大きくなったため、ある時点で選択可能なスケジューリング単位の数は減っている。

前者は相対的なオーバヘッド減少を意味するが、後者は並列性の低下につながるため、並列環境での実行では性能低下を引き起こす可能性がある。以下、スレッドのスケジューリング方式について考察し、同時に本研究で用いた手法の概略を述べる。

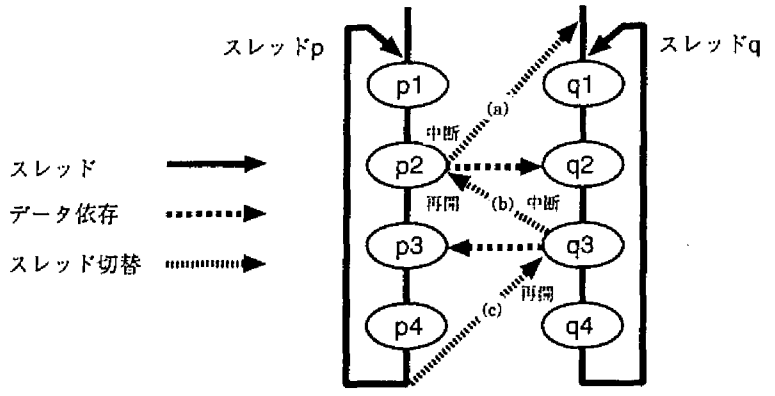


図 5.7: スレッドの resumption first スケジューリング

スレッドの resumption first スケジューリング

前項で述べたように、スレッド化によって、プロセス指向方式および resumption first 方式の欠点の多くは解決する。しかし、実行中のスレッドが終了するか中断したら他の実行可能なスレッドをスケジューリングするという単純な方式を用いると、並行プロセス間の切替がすべてゴールの中断を伴うという、プロセス指向の欠点が解消されない。

そこで、スレッド・スケジューリングでも resumption first 方式をとることを考える。つまり、新たに実行再開したゴールがあるかどうかを適当なタイミングでチェックし、もしあるなら、現在実行中のスレッドからそのゴールの所属するスレッドに、実行を切り替える。以下、これを resumption first 規則によるスレッド切替と呼ぶ。

ここで、スレッド d_iが中断し、スレッド d_jを実行中であるという状況を想定する。d_jのあるゴールの実行により d_iの中断原因となった変数が具体化されると、d_iは実行再開する (実行可能スレッドキューに戻される)。ここで、即座に (まだ残りのゴールを実行可能な)d_jから d_iに切り替えてしまうと、構造型データが完全に具体化されないという resumption first 方式の欠点が出てしまう。逆に、d_jが実行できなくなってから d_iに切り替えると、このときに必ず d_jのゴール中断を伴うことになる。

こうしたスレッドの切替のうち、少なくとも一方がループを含まない場合は、切替回数は定数回で済むため、大きな影響はない。したがって、問題になるのは図 5.7のように、それぞれループを含む 2 本のスレッドの間で相互依存を持つとき、すなわち、並行プロセス間での切替が起こる場合である。

本研究の手法では、ゴールの再帰呼出のときのみ resumption first 規則によるスレッド切替を適用することで、このような場合の中断回数を減らす。たとえば図 5.7の例で最初にスレッド p がスケジューリングされると、ゴール p1, p2 と実行が進み、p2 が中断してスレッド q に切り替わる (スレッド切替 (a))。次にゴール q1, q2 と実行が進んだ時点でゴール p2 が実行再開するが、この時点では resumption first 規則による切替は行われず、さらにゴール q3 へと進み、ここで中断してスレッド p に切り替わる (スレッド切替 (b))。その後、実行再開したゴール p2 に続き、ゴール p3 が実行され、q3 が実行再開するが、ここでもスレッド切替は行われず、そのまま p4 が実行される。次に、再帰呼出しにより p1 に戻る直前に初めて resumption first 規則による切替のチェックを行い、実行再開したスレッド q があるので、まだ実行可能なスレッド p から q に切り

替える (スレッド切替 (c))。以後、1 ループ毎にスレッド切替 (b), (c) が発生し、後者はゴールの中断を伴わない。このような形でスレッド単位の resumption first スケジューリングを行うことで、従来のゴール単位の resumption first 方式と同様に、並行プロセス切替のオーバヘッドを減らすことができる。

ただし、再帰呼出のすべてで resumption first 規則によるスレッド切替を行うと、一定長のリストを作る場合など、ループを用いて構造型データを生成する場合には、1 回ループした時点でスレッドが切り替わってしまい、具体化が不完全なまま並行プロセスが切り替わるという問題が解消しない。そこで、ループ内のゴール群と他のスレッドの間の依存を静的解析し、相互依存がある場合のみ、resumption first 規則による切替のチェックを行うことにする。

スレッドを並列実行する際の問題点

スレッドの resumption first スケジューリングを行うことで、単一ノードでの逐次実行に関しては、従来のスケジューリング方式で生じた問題は解決する。しかし、複数ノード上で並列実行を行う場合、本項の最初で述べたように、ゴールのスレッド化による並行性の低下が物理的な並列性を減少させ、実行速度が低下する可能性がある。以下、2.3.4 項で述べた分散メモリ版 KLIC の実行モデルを用いて説明するが、同種の問題は、他の分散メモリ並列計算機を対象とした KL1 の実装でも生じる。

単一ノードでの実行の場合、ゴールの実行順により枝刈り効率が変化する探索問題のような場合を除き、全体の仕事量はゴールの実行順序で変化することはない。したがって、スケジューリング処理のオーバヘッドがなるべく小さくなるようなゴール・スケジューリング方式を用いれば、それだけ実行時間は短くなる。しかし、複数ノードでの並列実行では、まだ全体の処理は終わっていないが、特定のノード N_i上では実行可能ゴールがまったくないという状況が起こり得る。この場合、他のノードから新たな実行ゴールが送られてくるか、他のノードでの具体化によって、現在中断しているゴールが実行再開するまでは、N_iはまったく仕事をしないアイドル状態になる。

従来のゴール・スケジューリング方式では、個々のゴールについて、中断しない限りスケジューリングの対象となる。これは余分なスケジューリングオーバヘッドを生む反面、実行可能な仕事が少しでも残っていれば、そのノードがアイドル状態に陥らないことを保証している。これに対し、本研究でのゴールのスレッド化は、相互依存により逐次化が不可能な部分以外は、なるべく同じスレッド内に含めようとする。これはスレッドの粒度をできるだけ大きくし、動的スケジューリングのオーバヘッドを最小化する戦略であるが、先行ゴールの中断に関わりなく実行可能なゴールも、同じスレッド内に配置してしまう。このため、あるスレッドが中断した場合、中断原因となったゴールの出力に依存しない、つまり実際はその時点で実行可能なゴールも、そのスレッドが実行再開するまではスケジューリングされないことがある。この結果、従来のスケジューリング方式では実行されたゴールであっても中断ゴールの再開後まで待たされてしまい、アイドル時間が増加する可能性がある。

たとえば、図 5.8のように、ゴール q1 がゴール p に依存しており、p が他のノードが具体化するデータを参照する場合を考える。これを、従来のゴール・スケジューリングで実行すると、read メッセージを送って p が中断した後、q が実行され、q1 が中断した後、q2 が実行される。この時点でまだ answer メッセージが返信されていなければ、アイドル状態になり、その後、answer メッセージが到着すると、p, q1 と実行される (図 5.8(a))。一方、同じプログラムをスレッド化すると、相互依存がないため全体が一本のスレッドになる。これを実行すると、p が中断した時点で残りのゴールすべてが実行されなくなり、即座にアイドル状態に入る。そして、answer メッセー

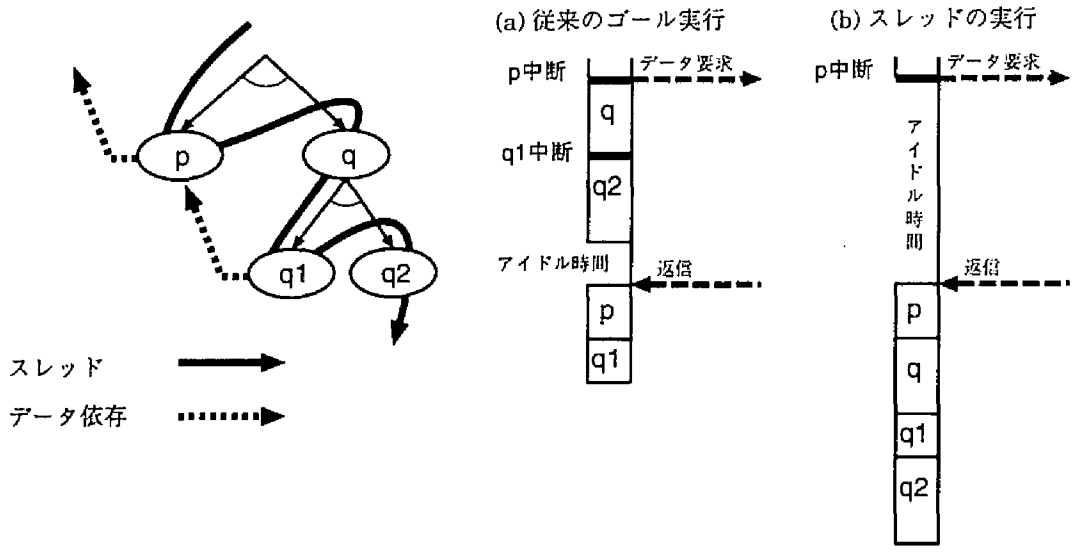


図 5.8: スレッドの並列実行によるアイドル時間の増加

ジが到着した時点で、p, q, q1, q2 と順に実行される (図 5.8(b))。この結果、中断処理時間などの減少分よりアイドル時間の増加が上回ると、スレッド化によってかえって実行速度が低下してしまうことになる。

この問題を解決するには、スレッド内のゴールと依存を持たないゴールはすべて別スレッドに分割するという方法がある。しかしこの場合は、スレッドの本数が増加するため、スケジューリング・オーバヘッドを減らすという本来の効果が発揮できない。後述するように、スレッドの粒度を動的に調整する方法も考えられるが、本研究では、このアイドル時間を減少させることのできる、新しいスレッド・スケジューリング方式を提案する。

reply first スケジューリング

スレッド化によるアイドル時間の増加を防ぐには、read メッセージを送ってから answer メッセージが到着するまでの時間を、できるだけ短くする必要がある。

read メッセージが到着した時点ですでに参照した変数が具体化されていれば、answer メッセージは直ちに返信される。この場合は、要求側が read メッセージ送信直後にアイドル状態に陥ったとしても、物理的な通信時間や answer メッセージの生成時間など、比較的小さな待ち時間で済む。しかし、参照変数が未具体化の場合、これが具体化されるまで answer メッセージの返信は延期される。ここで、具体化までの時間はまったく保証されていないため、この変数を具体化するゴールがなかなかスケジューリングされなかった場合、長時間にわたって返信が行われないことになる (図 5.9(a))。その間、データ要求側はずっとアイドル状態となり、並列性を減少させてしまう。

そこで、本研究ではスレッドのスケジューリング方式として、中断したスレッドのデータ要求に対し、値を生成するスレッドを優先的にスケジューリングするという、reply first 方式を提案する。本方式では、他ノードから read メッセージが到着したり、自ノード内のスレッドが未具体化変数を参照して中断したりすると、該当する変数を具体化するスレッドをスケジューリングす

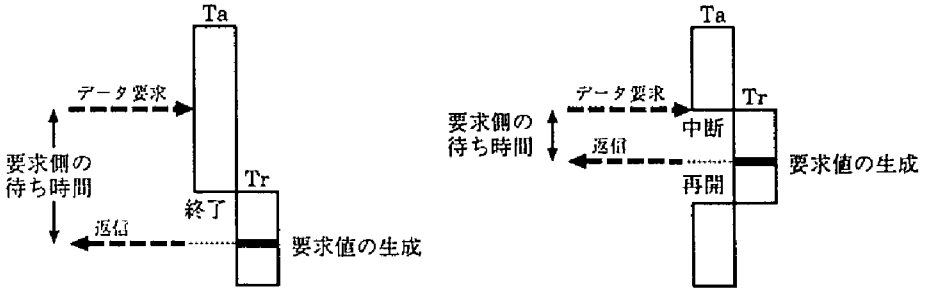


図 5.9: reply first スケジューリングによる応答時間の短縮

る。これによって、read メッセージに対して answer メッセージで応答するまでの時間が短縮され、アイドル時間を減少させることができる (図 5.9(b))。また、単一ノード上の実行についても、依存関係のあるスレッド群が次々に実行されることになるため、互いに独立性の高いスレッド群が複数ある場合、個々のグループを順次処理していく形になる。この結果、すぐには使われないデータを生成するスレッドが実行されることが少なくなり、メモリ効率が改善されることが期待できる。

このスケジューリング方式を実現するには、実行時に個々の変数について、それを具体化するスレッドがどれであるかという情報が必要である。したがって、本章の静的解析では、ゴール間の依存に加え、この情報も解析対象とする。

スレッドの動的粒度制御

本手法では、スレッドの reply first スケジューリングにより並列性低下の防止を図っているが、プログラムの性質によっては、やはりアイドル時間が無視できないものになる可能性がある。そこで、スレッドの粒度を下げることによって並行性を高め、従来のゴール・スケジューリング方式と同様に、アイドル時間を減少させる方法を考える。

この場合、スレッドの粒度を下げすぎるとゴール単位のスケジューリングと変わらなくなるため、並行性とスレッド粒度がトレードオフの関係になる。より柔軟な対応を可能にするためには、実行時の負荷や実際に生成されているスレッド数などに応じて、動的に粒度調整ができることが望ましい。

荒木らの研究 [54] では、ゴール自体を融合・インライン展開しているため、実行時に粒度を変えることができない。そこで、コード生成時に、粒度の高いコードと低いコードを二種類生成し、実行時にこれを動的に切り替えるという方法をとっている。

一方、本研究の手法では、スレッドとは実行順の定まったゴールの並びに過ぎない。したがって、スレッド内のゴールの実行順を入れ替えることはデッドロックの危険を招くが、任意のゴールを別のスレッドとして分離しても、なんら問題は生じない。そこで、実行時の状態を監視し、並行性が低すぎるようなら必要に応じて実行中のスレッドのゴールの一部を新スレッドとして分割することができる。ただし、先行ゴールに依存するゴールを分離しても意味がないため、動的なスレッド分割の対象にできるのは、先行ゴールに対し依存を持たないゴールである。そこで、解析時にこのようなゴールを区別しておき、実行時のノード状況によって、そのまま今のスレッド内で実行するか、新スレッドを生成するかを決定するようなコードを生成すれば良い。

5.4 静的解析

本節では、ゴールのスレッド化およびスレッド・スケジューリングのための静的解析手法について述べる。4.4 節と同様に、まずこれらの手法を実現するために必要な静的情報について考察し、その後、3.3 節の型解析手法を基に、必要な情報を解析する手法を説明する。

5.4.1 解析情報

ゴールのスレッド化に必要な情報

5.3.1 項で述べた依存のうち、制御依存はプログラムから容易に得ることができる。したがって、問題になるのはデータ依存の解析方法である。ゴール g_iが依存するのは、g_iが参照する変数 v_jの具体値を生成するゴール g_wと、g_wが生成した具体値と v_jを結び付ける同一化ゴール $g_{u1},\ldots,g_{um}$である。

本研究では、3.3 節で述べた型解析手法を拡張し、型に依存ゴールの情報を付加した解析を行う。本研究の型解析手法は、まず組込述語による型制約を求め、同一化ゴールやボディ・ヘッド間の同一化による伝搬を追跡する方法をとっている。ここで、組込述語による型制約のうち、具体化を行うものだけを考えると、それらが上記の g_wに相当するゴールになっている。また、こうした制約による具体値の型が伝搬して、参照を行う組込述語にたどり着くまでに通る同一化ゴールは、上記の $g_{u1},\ldots,g_{um}$に相当する。したがって、組込述語による型制約のうち、具体化を行うもののみを用いて 3.3 節と同様な型解析を行い、最初の組込述語、および伝搬途中の同一化ゴールをすべて記録していけば、各変数の型集合とともに、その変数が各型をとるまでに実行されなければならないゴールの集合、すなわち、各型が依存するゴールの集合が得られる。その結果、ゴール g_iの依存ゴールは、g_iが参照する型の依存ゴール集合の和を求めることで、得ることができる。

具体的な解析方法は、5.4.2, 5.4.3, 5.4.4 項で述べる。

スレッド・スケジューリングに必要な情報

ここでは、5.3.2 項で説明したスレッド・スケジューリング手法に必要な情報を検討する。

スレッドの resumption first スケジューリングを行うには、resumption first 規則による切替チェックを行うべきゴールを、区別しておく必要がある。5.3.2 項で述べたように本手法では、あるループがあって、ループ内のゴール群と他のスレッド間に相互依存があれば、そのループ末の再帰ゴールを呼び出す直前に切替チェックを行う。したがって、以下の解析が必要となる。

1. ループ構造、および、再帰ゴールの検出

2. ループ内ゴールとループ外との相互依存検出

KL1 では、ゴールの再帰呼出によるループが多用される。しかしこのループは、同じ述語の再帰呼出ゴールが複数箇所に出現するなど、複雑な構造を取り得る (図 5.10)。このため、手続き型言語の `for` ループのように開始位置と終了位置がプログラム中に明示されない。

そこで本論文では、ループ構造を次のように定義する。

ループの定義 あるプログラム中で、述語 p/n について p/n の定義節のボディゴール、あるいはその子孫ゴールにゴール p/n が出現する場合、このプログラムのゴール呼出木の中で、この述語

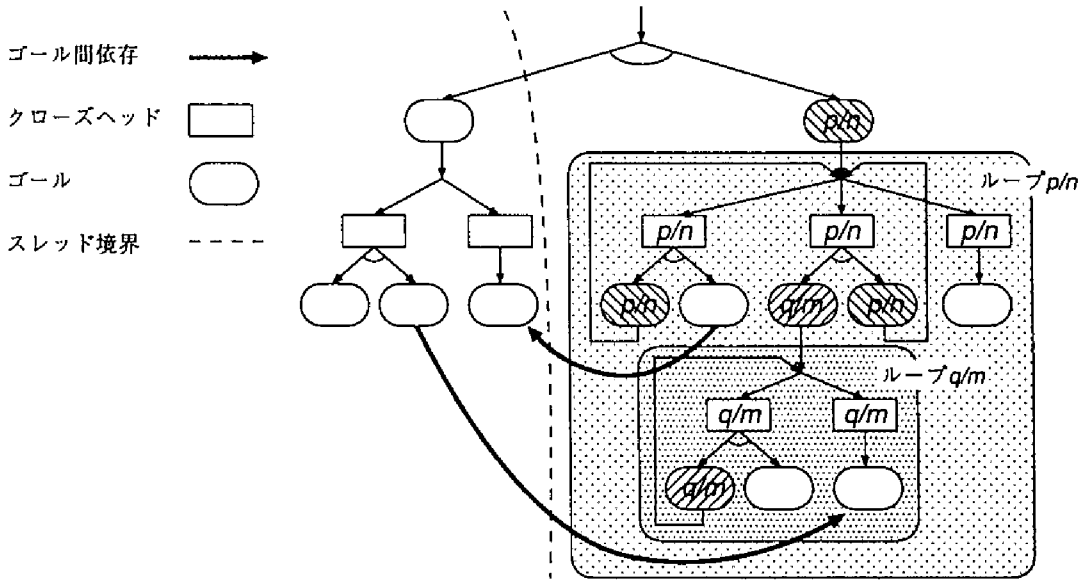


図 5.10: 再帰によるループ構造

を根とする部分木全体を、ループ p/n と呼ぶ。また、このときループ p/n 中に出現するゴール p/n を、再帰ゴールと呼ぶ。

たとえば、図 5.10では、述語 p/n および q/m がこの定義を満たすので、それぞれ図のようにループ p/n, q/n を形成する。

この定義により、プログラムの呼出木をトップダウンにたどることで、1. のループ構造と再帰ゴールは容易に検出できる。この情報と、各スレッドに含まれるゴールの情報、ならびにゴールのスレッド化のために求めたゴール間の依存情報と合わせることで、2. を得ることができる。具体的な解析方法は、5.4.6 項で述べる。

一方、スレッドの reply first スケジューリングを行うには、各変数について、具体化を行うスレッドが判っていれば良い。通常は参照と具体化はプログラム中の異なる節で行われるから、参照される変数と具体化される変数は異なっており、両変数が同一化されることによって生成された具体値が読み出される。したがってここでは、プログラム中で同一化される変数はすべて同じものとして考える。

同じスレッド内でしか具体化・参照されない変数は、そのスレッドの中断を引き起こさないから、具体化スレッドの情報は不要である。よって、以下の解析を行えば良い。

1. 同一化される変数の集合の解析

2. スレッド間の共有変数と非共有変数の区別

3. 共有変数について、自身を具体化するスレッドの解析

1. は、型解析で型波及を行う際に、変数間の同一化を記録しておけば良い。2. については、1. で求めた集合に、2 種類以上のスレッドの変数が含まれるかどうかで区別できる。また、3. は、基本的には具体化ゴールを含むスレッドを求めれば良いが、コード上は同じスレッドの実体が複数生成される場合など、具体化を行うスレッドが静的に決定しない場合がある。したがって、本研究

の reply first 方式では、実行時に具体化スレッドへのポインタを動的に変更していく方法をとる。この処理を行うための情報としては、後で述べるように、ユーザ定義ゴールの引数のモード情報があればよく、具体化スレッドを解析する必要はない。具体的な解析方法は、5.4.7 項で述べる。

解析情報の拡張

以上に述べた考察より、本章では解析情報を次のように拡張・追加する。

依存つき型集合 本章では変数 v_iの依存つき型集合を、$T(\mathrm{v}_i) = \{e_1, \ldots, e_n\}$ と定義する。各 e_j は型と依存ゴール集合の組 (t_j, D_j), 型変数と依存ゴール集合の組 (v_j, D_j), 不明要素$*$のいずれかである。

型 t_jについては 3 章の定義と同じであるが、本章ではこれに依存ゴールの集合 D_jを付加する。$g_l \in D_j$は v_iが型 t_jの値をとるのに依存するゴール、つまりこの値をとる前に実行される必要のあるゴールである。また 3 章と同様に、構造型データ型の引数として型変数 v_jを用いるが、これも依存ゴール集合 D_jとの組として考える。

以下の記述ではゴールに $\mathrm{p}/1^i$, $\mathrm{p}^j(\mathrm{X},\mathrm{Y})$, $\mathrm{X}=^k\mathrm{Y}$ のような形で番号を振り、依存ゴール集合をこのゴール番号の集合として表記する。

同一化集合 共有変数を解析するため、プログラム中で同一化されうる変数の集合 $U_1, \ldots, U_{umax}$ を考える。節選択時の非決定性により、プログラム中で記述された同一化はすべてが同時に起こるわけではないが、この同一化集合では、同一化の可能性がある変数の組合せはすべて同じ集合内に含んでいるとする。つまり、$U_i = \{\mathrm{v}_{i1}, \ldots, \mathrm{v}_{in}\}$ のとき、各 $\mathrm{v}_{ij}, \mathrm{v}_{ik}$は互いに同一化する可能性がある。

ユーザ定義述語の引数のモード ユーザ定義ゴール p/n の i 番目の引数のモードを、$m(p/n, i)$ で表す。モードは入力、出力をそれぞれ i,o とする。なお、ここで必要なのはゴール引数のモードのみなので、構造型データ型の引数のモードなどは考えない。

5.4.2 型解析の拡張

本項では、前項で定義した依存つき型集合と同一化集合、およびユーザ定義述語の引数のモードを得るため、3 章の型解析手法を拡張・変更する。

本章の解析では、4 章と同様に、対象プログラムは well-moded であり、同一性検査の意味を持つゴール=/2 は==/2 に書き換えられているものとする。また、議論を簡単にするために、ここでは対象プログラムはゴールの再帰呼出や、同じユーザ定義述語に対する複数の呼出を持たないこととする。再帰などを持つプログラムに対する解析については、5.4.3 項で議論する。

まず、次の演算および手続きを新たに定義する。

依存ゴールの追加演算

依存つき型集合 Tに依存ゴール集合 Dを追加する演算 $T_{AD} = AD(T, D)$ の結果は、以下を満たす最小の集合である。

1. 各 $(t_i, D_i) \in T$について、

(a) t_iがアトミックデータ型なら、
$T_{AD} \ni (t_i, D_i \cup D)$

(b) t_iが構造型データ型 $(s(f, n, (T_1, \ldots, T_n)), D_i)$ なら、
$T_{AD,k} = AD(T_k, D)$ とすると、$T_{AD} \ni (s(f, n, (T_{AD,1}, \ldots, T_{AD,n})), D_i \cup D)$

2. 各 $(v_i, D_i) \in T$について、
$T_{AD} \ni (v_i, D_i \cup D)$

3. $* \in T$なら $T_{AD} \ni *$

同一化集合への変数追加手続き

プログラム中の変数$\mathrm{v}_i, \mathrm{v}_j$が同一化されうることが判明したとき、次のように定義される手続き *entry_unifyset*$(\mathrm{v}_i, \mathrm{v}_j)$ によって、同一化集合 $U_1, \ldots, U_{umax}$を更新する。

1. $\mathrm{v}_i \in U_k \wedge \mathrm{v}_j \in U_l$のとき
$k \neq l$なら $U_k \leftarrow U_k \cup U_l$とし、$U_l$を削除。$k = l$なら更新しない。

2. $\mathrm{v}_i \in U_k \wedge \mathrm{v}_j \notin U_1 \wedge \ldots \wedge \mathrm{v}_j \notin U_{umax}$ のとき
$U_k \leftarrow U_k \cup \{\mathrm{v}_j\}$

$\mathrm{v}_i \notin U_1 \wedge \ldots \wedge \mathrm{v}_i \notin U_{umax} \wedge \mathrm{v}_j \in U_l$ のときも同様

3. $\mathrm{v}_i, \mathrm{v}_j \notin U_1 \wedge \ldots \wedge \mathrm{v}_i, \mathrm{v}_j, \notin U_{umax}$のとき
$umax \leftarrow umax + 1$, $U_{umax} = \{\mathrm{v}_i, \mathrm{v}_j\}$ とする。

また、依存つき型集合の OR 演算と同一化を、次のように定義する。

依存つき型集合の **OR** 演算

$T_{OR} = OR(T_A, T_B)$ とすると、T_{OR}は以下を満たす最小の集合である。

1. アトミックデータ型t_aについて

$(T_A \ni (t_a, D_A)) \wedge (T_B \ni (t_a, D_B))$

$\rightarrow T_{OR} \ni (t_a, D_A \cup D_B)$

$((T_A \ni (t_a, D_A)) \wedge (T_B \not\ni (t_a, D_B)))$

$\rightarrow T_{OR} \ni (t_a, D_A)$

$((T_A \ni (t_a, D_A)) \wedge (T_B \not\ni (t_a, D_B)))$ についても同様

2. 構造型データ型$t_{f/n}$について

$(T_A \ni (s(f, n, (T_{A,1}, \ldots, T_{A,n})), D_A)) \wedge (T_B \ni (s(f, n, (T_{B,1}, \ldots, T_{B,n})), D_B))$

$\rightarrow T_{OR,k} = OR(T_{A,k}, T_{B,k})$ とおくと
$T_{OR} \ni (s(f, n, (T_{OR,1}, \ldots, T_{OR,n})), D_A \cup D_B)$

$((T_A \ni (t_{f/n}, D_A)) \wedge (T_B \not\ni (t_{f/n}, D_B)))$

$\rightarrow T_{OR} \ni (t_{f/n}, D_A)$

$((T_A \not\ni (t_{f/n}, D_A)) \wedge (T_B \ni (t_{f/n}, D_B)))$ についても同様

3. 型変数 v_jについて $(T_A \ni (v_j, D_A)) \wedge (T_B \ni (v_j, D_B))$

$\rightarrow T_{OR} \ni (v_j, D_A \cup D_B)$

$((T_A \ni (v_j, D_A)) \wedge (T_B \not\ni (v_j, D_B)))$

$\rightarrow T_{OR} \ni (v_j, D_A)$

$((T_A \ni (v_j, D_A)) \wedge (T_B \not\ni (v_j, D_B)))$ についても同様

4. $(T_A \ni *) \vee (T_B \ni *) \rightarrow T_{OR} \ni *$

依存つき型集合の同一化

3.3.3 項の型解析アルゴリズムでは、型伝搬がデータフローの順方向・逆方向の双方向に生じる。そこで、型集合として与えられる型制約同士を融合するために、型集合の同一化を定義した。しかし、本章の型解析では、依存を追跡するために、データフローの順方向への型伝搬のみ行う。したがって、基本的に型集合間の同一化は必要ない。ただし、構造型データについては、その引数として出現する変数間の同一化を処理しなければならないから、引数に対する同一化処理を目的とした手続きが必要である。また、このとき同一化を生じたゴールを、依存ゴール集合に追加する必要がある。さらに、この種の同一化は型解析を行うまで判明しないから、判明した時点で同一化集合への追加処理を行わなければならない。そこで、3.3.3 項で定義した関数 $unify(T_A, T_B)$ の代わりに、本節では図 5.11のような手続き $unify(T_A, T_B, l)$ を定義する。ここで lは、この同一化を生じるゴールのゴール番号である。

このアルゴリズムでは、1, 2. で型変数間の型伝搬を行う。T_B中の組 (t_j, D_j) について、D_jが lを含む場合、その型は前回 $unify(T_A, T_B, l)$ を適用した際に、T_Aから T_Bへと伝搬したものである。したがって、lを含まない場合のみ、T_Bから T_Aへと型を伝搬させる。また、T_A, T_Bともに構造型データ型 $t_{f/n}$を含む場合は、3. で両者の対応する引数に対し、この手続きを再帰的に適用する。最後に 4. で、生じ得る型変数間の同一化を、同一化集合に登録する。

依存つきの型解析

各 $\mathrm{v}_i \in V$の依存つき型集合 $T(\mathrm{v}_i)$ は、3.3.3 項の型解析アルゴリズムに対し、次のような変更を加えることで求められる。

1. 最初に組込述語より型制約を求める際に出力引数の型制約のみ求め、得られた型の依存ゴール集合の初期値として、その出力引数を持つゴール番号を要素とする集合を与える。以後、データフローの順方向のみの型伝搬を行う。

2. クローズヘッド・ボディゴール間の型伝搬では、3.3.3 項の OR 演算の代わりに、本項で定義した OR 演算を用いる。

1. **forall** $(v_i, D_i) \in T_A$ {
   if $\exists t_j((t_j, D_j) \in T_B \wedge (D_j \not\ni l))$ {
   $T(\mathrm{v}_i) \leftarrow AD(T_B, \{l\})$
   }
   else if $\exists v_k((v_k, D_k) \in T_B \wedge D_k \not\ni l) \wedge \neg\exists t_m((t_m, D_m) \in T(\mathrm{v}_k) \wedge D_m \not\ni l)$
   $\wedge \neg\exists v_m((v_m, D_m) \in T(\mathrm{v}_k) \wedge D_m \not\ni l)$ {
   $T(\mathrm{v}_i) \leftarrow AD(T_B, \{l\})$
   }
   }
2. **forall** $(v_i, D_i) \in T_B$ {
   if $\exists t_j((t_j, D_j) \in T_A \wedge (D_j \not\ni l))$ {
   $T(\mathrm{v}_i) \leftarrow AD(T_A, \{l\})$
   }
   else if $\exists v_k((v_k, D_k) \in T_A \wedge D_k \not\ni l) \wedge \neg\exists t_m((t_m, D_m) \in T(\mathrm{v}_k) \wedge D_m \not\ni l)$
   $\wedge \neg\exists v_m((v_m, D_m) \in T(\mathrm{v}_k) \wedge D_m \not\ni l)$ {
   $T(\mathrm{v}_i) \leftarrow AD(T_A, \{l\})$
   }
   }
3. **forall** $(t_i, D_i) \in T_A$ {
   if $t_i = s(f, n, (T_{A,1}, \ldots, T_{A,n})) \wedge s(f, n, (T_{B,1}, \ldots, T_{B,n})) \in T_B$ {
   for $i = 1$ **to** n {
   unify$(T_{A,i}, T_{B,i}, l)$
   }
   }
   }
4. **forall** $(v_i, D_i) \in T_A$ {
   forall $(v_j, D_j) \in T_B$ {
   entry_unifyset$(\mathrm{v}_i, \mathrm{v}_j)$
   }
   }

図 5.11: 依存つき型集合に対する手続き $unify(T_A, T_B, l)$ のアルゴリズム

```
1 main :- p[1](X), X=[2]Y, q[3](Y).
2 p(A) :- A=[4]1.
3 p(A) :- A=[5]a.
4 q(B) :- integer[6](B) | true.
5 q(B) :- atom[7](B) | true.
```

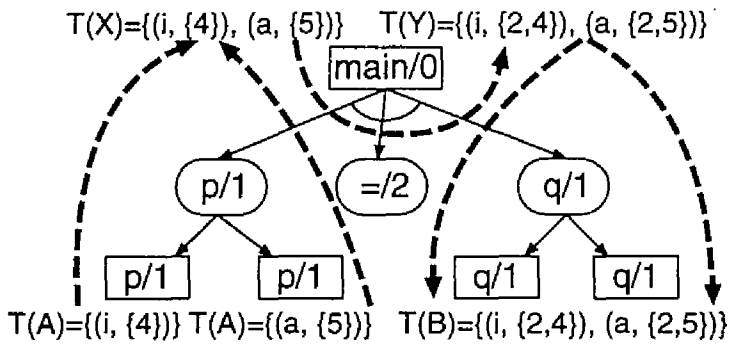


図 5.12: 依存つき型集合の解析例 (1)

3. 同一化ゴール$=/2^l$により型集合が伝搬する際には、依存ゴールの追加演算 $AD(T, \{l\})$ によって、伝搬先の型集合 Tに lを依存ゴールとして加える。

4. 組込述語の入力引数については基本的に考慮しなくてよいが、クローズヘッドの具体値引数やガードゴール==/2 によって、構造型データとの同一性検査が行われる場合には、引数として出現する変数との同一化が生じる可能性がある。そこで、このような同一化検査については手続き *unify* を適用する。

依存つき型解析の例を、図 5.12, 5.13に示す。

図 5.12の例では、まず節 2,3 においてゴール 4,5 の出力引数である変数 A の型制約が得られ、依存ゴール集合の初期要素としてそれぞれゴール 4,5 が与えられる。次に、ゴール p/1 のクローズヘッド・ボディゴール間同一化によって、変数 X にこれらの型制約が波及し、さらにゴール 2 によって変数 Y にも波及する。このとき T(Y) は、T(X) の依存ゴール集合に経由した同一化ゴール $=/2^2$を加えたものとなる。その後、ゴール q/1 のクローズヘッド・ボディゴール間同一化によって節 4,5 の T(B) が決定し、解析が終了する。

また、図 5.13のプログラムでは、構造型データを介した双方向のデータフローが生じている。このプログラムの解析では、まず節 2,4 においてそれぞれ変数 A,D の型制約が得られる。次に、T(A) が T(X) → T(Y) → T(B) と順に伝搬する。ここで、ゴール==/2^7は構造型データとの同一性検査であるから、前記の変更点 4 より手続き $unify(T(\mathtt{B}), \{(s(\mathtt{f}, 1, (\{(D, \{7\})\})), \{7\})\}, 7)$ を適用する。この結果、T(D) が T(C) に伝播し、後者の依存ゴール集合は、T(D) の依存ゴール集合 {8} と T(B) 中の型変数 Cの依存ゴール集合 {2, 4} の和に、この伝搬を生じたゴール番号 7 を加えたものになる。そして最後に p1/1 のクローズヘッド・ボディゴール間同一化によって T(C) が T(E) に波及し、解析が終了する。

```
1 main :- p[1](X), X=[2]Y, q[3](Y).
2 p(A) :- A=[4]f(C), p1[5](C).
3 p1(E) :- integer[6](E) | true.
4 q(B) :- B==[7]f(D) | D=[8]1.
```

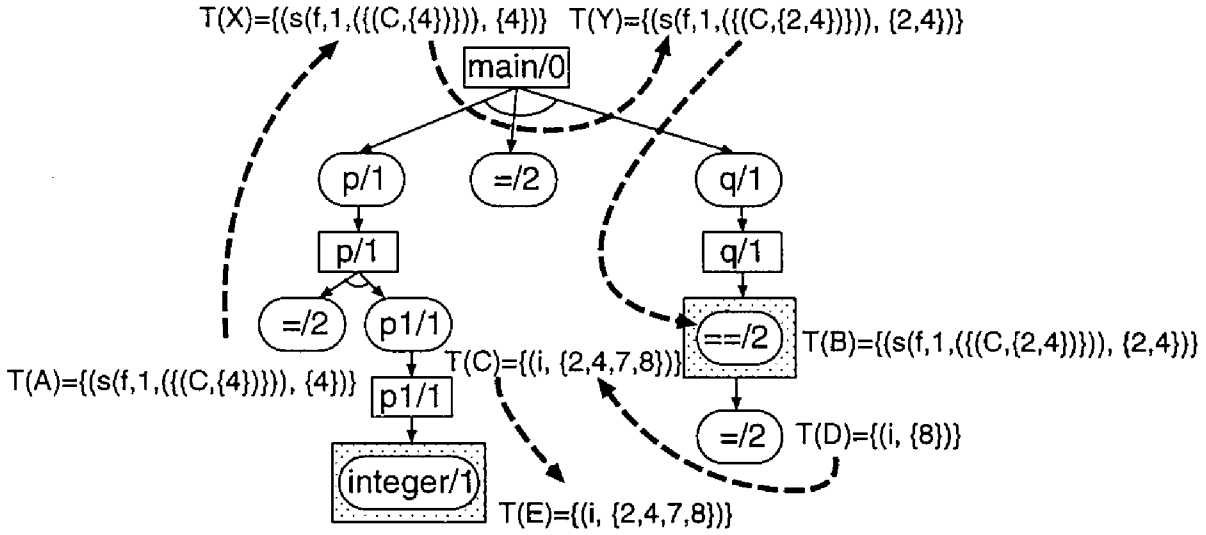


図 5.13: 依存つき型集合の解析例 (2)

5.4.3 共有述語を含むプログラムの解析

5.4.2 項では、対象プログラムが同じユーザ定義述語に対する複数の呼出を持たないことを仮定した。本項では、この仮定を取り除いた場合の問題点を考察し、一般の KL1 プログラムを扱えるように解析手法を拡張する。

同一述語の複数呼出による問題点

解析対象となるプログラムが、同じユーザ定義述語に対し複数の呼出 (ゴール) を持つとき、5.4.2 項の解析手法は、不正確な依存を導き出す可能性がある。

まず、プログラムが再帰ゴールによるループ構造を持つとき、実際には存在しない見掛けの依存が得られる場合がある。これは、本解析手法ではループの世代を考慮していないため、前世代のゴールへの依存が同世代の同じゴールへの依存と同様に扱われてしまうからである。たとえば図 5.14のプログラムに 5.4.2 項の解析を適用すると、add/3 と subtract/3 が相互依存を持つという結果が得られる。このため、これをスレッド化すると、両ゴールが別スレッドに分割されてしまう。しかし、実際は add/3 の入力引数 N はループの前世代の subtract/3 の出力引数 N2 の値を参照しており、相互依存は存在しない。したがって、このプログラムは 1 スレッドに逐次化可能である。

また、ループ構造を持たないプログラムでも、ゴールの呼出木中に、複数の親ゴールに共有された部分木が存在するとき、やはり見掛け上の依存を生じることがある。5.4.2 項の解析手法は、各ボディゴール p/n の依存を述語 p/n の依存として融合するため、各ボディゴールがそれらすべ

```
main :- X=[1]0, p[2](X).
p(N) :- N<[3]10 | add[4](N,3,N1), subtract[5](N1,2,N2), p[6](N2).
p(N) :- N=[7]10 | true.
```

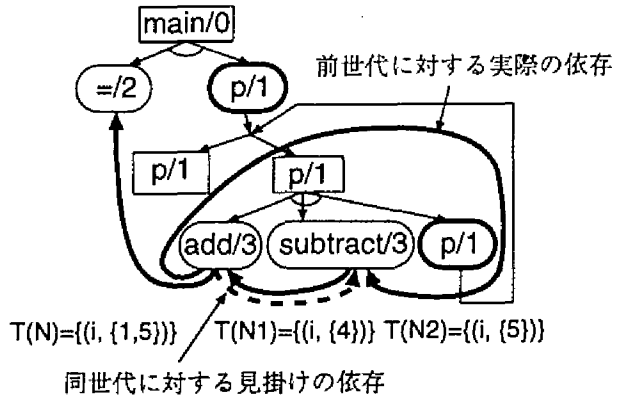


図 5.14: ループ構造による見掛けの依存

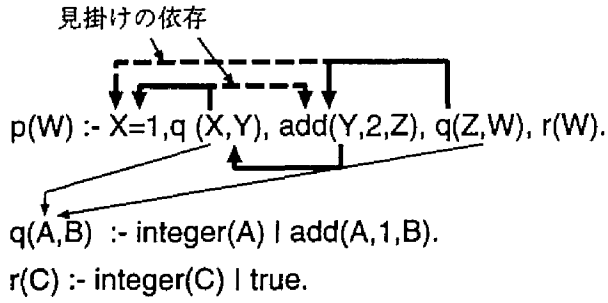


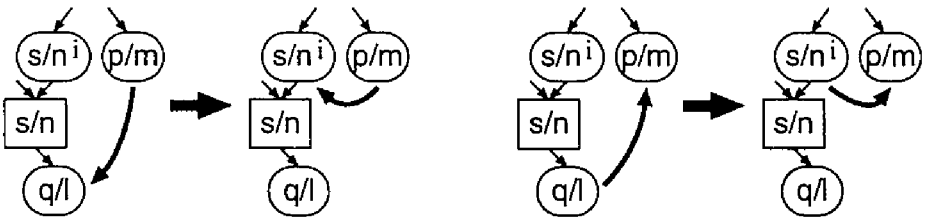
図 5.15: 共有述語による見掛けの依存

ての依存を持つように見えてしまう。たとえば図 5.15の例では、ボディゴール q/2 が 2 個存在する。これらはそれぞれゴール=/2, add/3 に依存するが、両者の依存が融合されてしまうため、ともに=/2 と add/3 の双方に依存するように見える。このため、最初の q/2 と add/3 の間に見掛け上の相互依存が生じ、この節は 2 スレッドに分割されてしまう。

本論文では、プログラム中にユーザ定義ゴール p/n が複数存在するとき、これらを共有ゴール、述語 p/n を共有述語と呼ぶ。ゴール r/m の再帰呼出によるループ構造は、ループの開始ゴール r/m と 1 個以上の再帰ゴール r/m から成るため、r/m も共有述語の一種である。したがって、共有述語に対して正しい解析ができれば、上記の問題を解決することができる。

共有述語に対する見掛けの依存の解消

ここで、b_iの子孫ゴールの集合を $Des(b_i)$ と表記することとして、共有述語 s/n、その呼出の一つであるゴール s/n^i、および 2 個のゴール $q/l \in Des(s/n^i)$, $p/m \notin Des(s/n^i)$ を考える。本研究のスレッド化では、兄弟ゴール間の順序決定しか行わないから、p/m から q/lへの依存を p/m



(a) 共有ゴールへの依存　　(b) 共有ゴールからの依存

図 5.16: 共有述語の依存

から q/lの先祖ゴールである s/n^iへの依存に置き換えても、ゴール間の順序制約は変わらない (図 5.16(a))。同様に q/lから p/m への依存は、s/n^iから p/m への依存に置き換えることができる (図 5.16(b))。

この置き換えを実現するには、5.4.2 項の解析手法に、以下の変更を加えれば良い。

1. ボディゴールからクローズヘッドへの型伝搬を行う際に、ボディゴール引数の型集合 $T(\mathrm{v}_b)$ より T_{BH}を生成する。このとき型伝搬を生じた述語が共有述語 s/n なら、$T(\mathrm{v}_b)$ に含まれる依存ゴール集合はボディゴール s/n^iの子孫ゴールに伝搬させず、ゴール s/n^iの依存とする。これによって、図 5.16(b) のような置き換えが実現できる。

   しかしながら、単純にゴール s/n^iの依存に加えてしまうと、実際はその型が s/n^iの子孫ゴールで参照されない場合、不正確な依存を生じてしまう。そこで、依存集合の要素として (i, D_k) という形を導入する。i, D_kはそれぞれ共有ゴールのゴール番号および依存ゴール集合であり、この要素を含む依存ゴール集合と組になった型が参照される場合に、共有ゴール g_iから D_kへの依存を生じることを表す。

2. クローズヘッドからボディゴールへの型伝搬を行う際に、ヘッド引数の型集合より T_{HB}を生成する。このとき型伝搬を生じた述語が共有述語 s/n なら、これをボディゴール s/n^iの引数に適用する際に s/n^iの子孫ゴールを依存ゴール集合 D_kから取り除き、代わりの依存ゴール集合として、共有ゴールのゴール番号 i を要素とする集合を与える。これによって、図 5.16(a) のような置き換えが実現できる。

   ただし、ここで D_kが 1. で導入した (i, D'_k) の形の要素を含むことがある。これは、共有ゴール s/n^iの入力引数として 1. の置き換えを行った型と依存ゴール集合の組が、共有ゴールを素通りして出力引数の型集合中に現れる場合に生じる。このときは、D'_k中のゴールは s/n^iの子孫ではないから、D'_kも伝搬先の依存に加える必要がある。

以上の変更により、たとえば図 5.14のプログラムは、見掛け上の依存が解消され、図 5.17のような解析結果を得ることができる。

また、この変更によって再帰ゴールの子孫ゴールの依存は、再帰ゴールの依存に置き換えられる。したがって以下に述べる処理では、再帰ゴールは単に子孫ゴールを持たないゴールとして扱えばよい。これによって、再帰構造によりアルゴリズムが無限ループに陥るのを防ぐことができる。

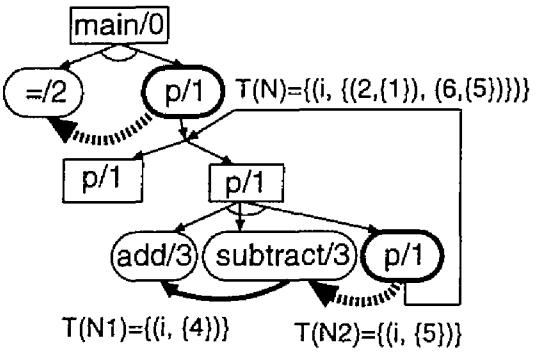


図 5.17: ループ構造による見掛けの依存の解消

依存つきの型解析アルゴリズム

上記の変更を加えた、依存つき型解析アルゴリズムの詳細を、図 5.18, 5.19に示す。なお、3.3.3 項で用いた記号の他に、ガードゴール、ボディゴールの集合をそれぞれ G_g, G_b、任意の構造型データを s_mで表している。また、型伝搬の逆流を防ぐため、更新プール Nには、変数とともに更新を生じたゴール番号を組にして格納する。

このアルゴリズムでは、1, 2. で図 3.2のアルゴリズムと同様の初期化を行い、3, 4, 5. で同一化集合を生成する。続いて 6. で具体化ゴールより型制約を求め、7. でボディゴールの引数として現れる具体値を更新プールにいれる。データフロー方向への伝搬のみを行うために、ここでは具体化を行うゴールのみを処理する。

8. では図 3.2の 5. と同様に、型集合の更新がなくなるまで型伝搬を繰り返す。(a) では図 3.2と同様に節内の型伝搬を行うが、伝搬した型にはそのゴール番号を追加する。本項の型解析では、値を参照する側のゴールは基本的に考慮する必要がないが、構造型データと変数の同一化については、引数として出現する変数について同一化が生じる可能性がある。そこで (b),(c) では、ガード部での構造型データ型との同一化検査について、手続き *unify* を適用する。また、(d),(e) では、図 3.2(b),(c) と同様に、クローズヘッド・ボディゴール間の型伝搬を処理する。このとき、型伝搬の行われる向きにこの述語の引数モードを決定する。ここで、クローズヘッドの引数に具体値 b_h が現れる場合はガードゴールと同様の検査を意味するため、手続き *unify* を適用する。また、共有述語に対しては前記したように依存ゴール集合の処理を行う。

5.4.4 ゴール間の依存解析

組込述語については、各引数の入出力方向と取り得る型が決まっている。これを利用して型解析の結果より、各ゴールの直接データ依存集合を得ることができる。

まず、組込述語 b/n の i 番目の引数が入力かつ許容される型の集合が $T_{b/n,i} = \{t'_1, \ldots, t'_m\}$ であり、ゴール b/n の i 番目の引数が変数 v_jである場合、v_jの具体値が依存するゴールの集合は、図 5.20に示す関数 $dd\_arg(T_{b/n,i}, \mathrm{v}_j)$ によって求めることができる。この関数は、得られた型集合 $T(\mathrm{v}_j)$ より、その引数の位置に取り得る型の依存ゴール集合の和を求めている。型変数 v_kについては、2. で再帰的に $T(\mathrm{v}_k)$ について調べている。また、v_jが $T(\mathrm{v}_k)$ に含まれる型を取るためには D_k中のゴールも実行される必要があるため、これらも D_dに加える。

次に、ゴール gの直接データ依存集合 $D_d(g)$ を求める関数 $dd\_goal(g)$ のアルゴリズムを、図 5.21

1. $N \leftarrow \emptyset$ /* 更新変数集合を初期化 */
2. forall $\mathrm{v}_j \in V$ {
   $T(\mathrm{v}_j) \leftarrow \{*\}$ /* 各変数の型集合を初期化 */
   }
3. $umax \leftarrow 0$
4. forall $=^i(\mathrm{v}_j, \mathrm{v}_k) \in G$ {
   $entry\_unifyset(\mathrm{v}_j, \mathrm{v}_k)$
   }
5. forall $p/n, m$ {
   forall $\mathrm{v}_j \in H_{p/n,m}$ {
   forall $\mathrm{v}_k \in B_{p/n,m}$ {
   $entry\_unifyset(\mathrm{v}_j, \mathrm{v}_k)$
   }
   }
   }
6. forall $=^i(\mathrm{v}_j, \mathrm{b}_k) \in G$ {
   $T(\mathrm{v}_j) \leftarrow \{(t(b_k), \{i\})\}$, $N \leftarrow N \cup \{(\mathrm{v}_j, i)\}$
   =/2 以外の組込述語についても、同様に出力変数の型集合を作成
   }
7. forall $b_b \in B_{p/n,m}$ {
   $N \leftarrow N \cup \{(b_b, i)\}$ (g_iは b_bを第 m 引数に持つゴール p/n)
   }
8. while $N \neq \emptyset$ {
   $(x, i) \leftarrow N$の要素, $N \leftarrow N \setminus \{(x, i)\}$
   if $x \in V$ {
(a) forall $=^l(x, \mathrm{v}_k) \in G$ {
   $T(\mathrm{v}_k) \leftarrow AD(T(x), \{l\})$, $T(\mathrm{v}_k)$ が変更されたら $N \leftarrow N \cup \{(\mathrm{v}_k, l)\}$
   }
(b) forall $==^l(x, s_m) \in G$ ($l \neq i$) {
   $unify(\{(t(s_m), \{l\})\}, T(x), l)$
   変更のあった各 $T(\mathrm{v}_u)$ について、それぞれ $N \leftarrow N \cup \{(\mathrm{v}_u, l)\}$
   }
(c) forall $==^l(\mathrm{v}_k, s_m) \in G$ ($l \neq i$, s_m中に x が現れるなら) {
   $unify(\{(t(s_m), \{l\})\}, T(\mathrm{v}_k), \{l\})$
   変更のあった各 $T(\mathrm{v}_u)$ について、それぞれ $N \leftarrow N \cup \{(\mathrm{v}_u, l)\}$
   }
   }

図 5.18: 依存つきの型解析アルゴリズム (1)

```
(d)  forall H_{p/n,m}  s.t.  x ∈ H_{p/n,m} ∧ x ∈ V ∧ g_i ≠ p/n {
       T_HB ← ∅
       m(p/n, m) ← o
       forall v_h ∈ H_{p/n,m} {
         T_HB ← OR(T_HB, T(v_h))
       }
       forall v_b ∈ B_{p/n,m} {
         if p/n が共有述語 {
           forall D_k  (D_k は T_HB 内の依存ゴール集合) {
             D ← {s}  (s は v_b を第 m 引数に持つゴール p/n のゴール番号)
             forall (s, D'_k) ∈ D_k {
               D ← D ∪ D'_k
             }
             D_k ← D
           }
         }
         T(v_b) ← T_HB
         T(v_b) が変更されたら N ← N ∪ {(v_b, l)}  (g_l は v_b を第 m 引数に持つゴール p/n)
       }
     }
(e)  forall B_{p/n,m}  s.t.  x ∈ H_{p/n,m} ∧ g_i ≠ p/n {
       T_BH ← ∅
       m(p/n, m) ← i
       forall v_b, b_b ∈ B_{p/n,m} {
         if p/n が共有述語 {
           forall D_k  (D_k は T_HB 内の依存ゴール集合) {
             D_k ← {(s, D_k)}  (s は v_b を第 m 引数に持つゴール p/n のゴール番号)
           }
         }
         T_BH ← OR(T_BH, T(v_b))
         T_BH ← OR(T_BH, {(t(b_b), {l})})  (g_l は b_b を第 m 引数に持つゴール p/n)
       }
       forall v_h, b_h ∈ H_{p/n,m} {
         T(v_h) ← T_BH
         unify({(t(b_h), ∅)}, T_BH, l)
         変更のあった各 T(v_u) について、それぞれ N ← N ∪ {(v_u, l)}
         (g_l は x を第 m 引数に持つゴール p/n)
       }
     }
(f)  forall v_k ∈ V{
       if T(v_k) ∋ (v_k, D_k) {
         T(v_k) から (v_k, D_k) を除く
       }
     }
   }
```

図 5.19: 依存つきの型解析アルゴリズム (2)

1. $D_d \leftarrow \emptyset$ /* 返り値の初期化 */
2. forall $(t_k, D_k) \in T(\mathrm{v}_j)$ {
   if $t_k \in T_{b/n,i}$ {
   $D_d \leftarrow D_d \cup D_k$
   }
   }
3. forall $(v_k, D_k) \in T(\mathrm{v}_j)$ {
   $D_d \leftarrow D_d \cup dd\_arg(T_{b/n,i}, \mathrm{v}_k)$
   D_dが更新されたら、$D_d \leftarrow D_d \cup D_k$
   }

図 5.20: 関数 $dd\_arg(T_{b/n,i}, \mathrm{v}_j)$ のアルゴリズム

に示す。組込述語の場合、入力引数として出現する各変数について、関数 dd_arg により得られる依存ゴールの和を取ればよい。ユーザ定義述語の場合、その述語の各定義節について、まず各ガードゴールの依存ゴールを求め、それらの和を取ればよい。さらに、得られた依存ゴール集合に (s_l, D_l) の形式の要素が含まれる場合は、共有ゴール s_lが D_lにデータ依存することを表している。そこで、ゴール g_{s_l}の直接データ依存集合 $D_d(g_{s_l})$ に D_lを加え、関数の返り値からはこの要素を取り除く。

本研究のスレッド化では、各節内のボディゴール $b_1, \ldots, b_n$間で順序付けを行う。これに必要なゴール間の順序制約は、上記の方法で得られた直接データ依存と、5.3.1 項の定義から得られる直接制御依存 $D_c(g_i)$ より得ることができる。任意のボディゴール b_iについて、b_iに先行して実行すべきゴールの集合を $Prec(b_i)$、b_iの子孫ゴールの集合を $Des(b_i)$ と表記すると、$Prec(b_i)$ は図 5.22 のアルゴリズムを、呼出木上で葉のゴールから順に、ボトムアップに適用していくことで求められる。

このアルゴリズムでは、まず 1. で $Prec(b_i)$ を直接データ依存で初期化する。直接制御依存は同じ節内のボディゴール間の依存には影響しないので、ここでは考える必要がない。続いて間接依存を展開する。b_iがデータ依存するゴールの制御依存ゴールが b_iの兄弟ゴールである可能性があるから、依存ゴールの制御依存ゴールは兄弟ゴール間の順序を制約する可能性がある。したがって、2. のように、間接データ依存、間接制御依存ともに $Prec(b_i)$ に加える。また、間接依存ゴールの依存も b_iの依存となるから、新たに発見された依存ゴールは未展開ゴールのプール Nに追加し、再帰的に間接依存の展開を行う。さらに、本手法のスレッド化では b_iが実行を開始すると、同スレッド内の b_iの子孫ゴールはそのまま続けて実行される。そのため、b_iの子孫ゴールの先行ゴールは、やはり b_iに先行して実行しておく必要がある。そこで 3. のように、子孫ゴールの先行ゴールを $Prec(b_i)$ に加える。

任意の $b_j \in Prec(b_i)$ について、b_jが b_iに先行して実行されなければならない、という順序制約が存在する。これを、$prec(b_j, b_i)$ と表記する。

1. $D_d \leftarrow \emptyset$ /* 返り値の初期化 */
2. if gが組込述語 b/n {
   for $i = 1$ to n {
   if b/n の i 番目の引数が入力かつ取り得る型の集合が $T_{b/n,i}$∧
   ゴール gの i 番目の引数が変数 v_j {
   $D_d \leftarrow D_d \cup dd\_arg(T_{b/n,i}, \mathrm{v}_j)$
   }
   }
   }
3. else if gがユーザ定義述語 u/n {
   forall $c_i \in C_{u/n}$ {
   c_iのガードゴールを $g_{g1}, \ldots, g_{gm_i}$とすると、
   $D_d \leftarrow D_d \cup dd\_goal(g_{g1}) \cup \ldots \cup dd\_goal(g_{gm_i})$
   }
   }
4. forall $(s_l, D_l) \in D_d$ {
   $D_d(g_{s_l}) \leftarrow D_d(g_{s_l}) \cup D_l$
   $D_d \leftarrow D_d \setminus D_l$
   }

図 5.21: 関数 $dd\_goal(g)$ のアルゴリズム

1. $Prec(b_i) \leftarrow D_d(b_i)$, $N \leftarrow Prec(b_i)$
2. while $N \neq \emptyset$ {
   $b_k \leftarrow N$の要素, $N \leftarrow N \setminus \{b_k\}$
   $N \leftarrow N \cup ((D_c(b_k) \cup D_d(b_k)) \setminus Prec(b_i))$
   $Prec(b_i) \leftarrow Prec(b_i) \cup D_c(b_k) \cup D_d(b_k)$
   }
3. forall $b_l \in Des(b_i)$ {
   $Prec(b_i) \leftarrow Prec(b_i) \cup Prec(b_l)$
   }

図 5.22: 先行ゴール集合 $Prec(b_i)$ を求めるアルゴリズム

1. $pred \leftarrow$ main/0
2. forall $c_{pred} \in C_{pred}$ {

(a) forall c_{pred}のボディゴール b_i(述語 p/n とする){
p/n がユーザ定義述語なら $pred \leftarrow p/n$ として 2. を再帰呼出
$Prec(b_i)$ を求める
}

(b) $gmax \leftarrow 0$
forall c_{pred}のボディゴール b_i {
if $\exists k\ (\neg\exists j\ (prec(b_{k,j}, b_i) \wedge prec(b_i, b_{k,j})))$ {
$G_k \leftarrow G_k \cup \{g_i\}$
}
else {
$gmax \leftarrow gmax + 1,\ G_{gmax} \leftarrow \{b_i\}$
}
}

(c) for $k = 1$ to $gmax$ {
$T_k \leftarrow \emptyset$
forall $b_{k,i} \in G_k$ {
$\neg\exists j'\ (j' < j \wedge prec(b_{k,i}, b'_{k,j'})) \wedge \neg\exists j''\ (j'' \geq j \wedge prec(b'_{k,j''}, b_{k,i}))$ を満たす jが必ず存在するので、
$T_k \leftarrow \{\ldots, b'_{k,j-1}, b_{k,i}, b'_{k,j}, \ldots\}$
}
}
}

図 5.23: スレッド化アルゴリズム

5.4.5 プログラムのスレッド化

プログラムのスレッド化を行うには、型解析の結果より各節についてボディゴール間の順序制約 $prec(b_i, b_j)$ を求め、ボディゴール間の順序とスレッド分割を行う。具体的なアルゴリズムを図 5.23に示す。

2(a) では、節内の各ゴール b_iについて $Prec(b_i)$ を求める。図 5.22より、この計算には子孫ゴールの $Prec(b_i)$ が必要であるから、まず子ゴールを再帰的に処理するというボトムアップの計算を行う。次に、2(b) でゴールのスレッド分割を行う。各スレッドに含まれるゴールの集合を $G_1, \ldots, G_{gmax}$ で表すと、相互依存を持つゴールが異なる集合に含まれるようにすることで、各 G_kは半順序集合となる。最後に、各スレッド内のゴールの順序づけを行う。各 G_k内のゴールを順序づけしたものを順序集合 $T_k = \{b'_{k,1}, \ldots, b'_{k,m_k}\}$ で表すことにすると、半順序集合の要素は必ず順序づけ可能であるから、2(c) のようにして T_kが求められる。

こうして得られた順序集合の集まり $T_1, \ldots, T_{gmax}$より、次のようにして節内ゴールのスレッド

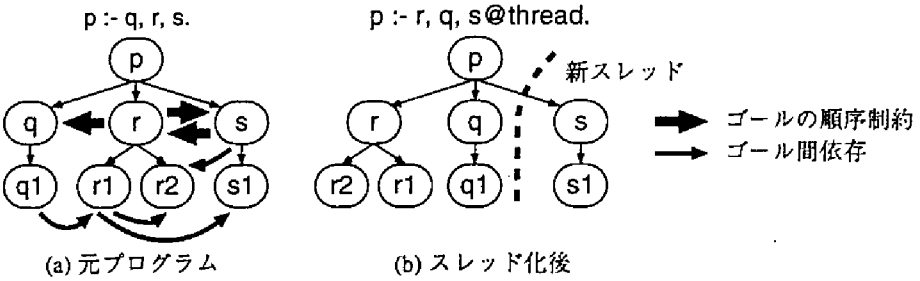


図 5.24: 節のスレッド化

化を行う。

1. $b'_{1,1},\ldots,b'_{1,m_1},\ldots,b'_{gmax,1},\ldots,b'_{gmax,m_{gmax}}$ の順に、ゴールを並べ替える。

2. T_1 を除く各順序集合の先頭ゴール $b'_{2,1},\ldots,b'_{gmax,1}$ に、プラグマ@threadを付加する。@thread は本研究で新設したプラグマで、新スレッドの開始ゴールを意味する。

たとえば、図 5.24(a) の例では、ゴール r, q はこの順に順序づけ可能であるが、ゴール s は r と相互依存しているので、r, q と同じ順序集合には入れられない。したがって、$T_1 = \{\mathtt{r},\mathtt{q}\}$, $T_2 = \{\mathtt{s}\}$ となり、この節をスレッド化した結果は図 5.24(b) のようになる。この結果、r, q は親ゴール p と同じスレッド内で順に実行されるが、s は新スレッドが生成され、そこで実行される。

5.4.6 ループ内外の相互依存解析

5.3.2 項で述べたように、本手法ではあるスレッドがループを含む場合、ループ内と他のスレッド間に相互依存がある場合のみ、再帰ゴール呼出前にスレッド切替チェックを行う。このため、ループ内ゴールと他のスレッド内のゴールの間に、相互依存があるかどうかの解析が必要となる。

ここで、ループ p/n に含まれるゴールの集合 $L_{p/n}$、および、スレッド q/m に含まれるゴールの集合 $T_{q/m}$ を考える。ループ p/n を含むスレッドを q/m とすると、q/m 以外のあるスレッド r/l について、

1. $g_i \in L_{p/n}, g_j \in T_{r/l}$ かつ $g_j \in Prec(g_i)$

2. $g_{i'} \in L_{p/n}, g_{j'} \in T_{r/l}$ かつ $g_{i'} \in Prec(g_{j'})$

となるゴール $g_i, g_j, g_{i'}, g_{j'}$ が存在するとき、上記の相互依存を生じる。しかし、5.4.3 項で述べたように、ループ p/n の開始ゴールを $g_s(=p/n)$ としたとき、ループ内のゴール g_i から他スレッドへの依存、および他スレッドから g_i への依存は、すべて g_s からの依存や g_s への依存に置き換えられている。したがって実際には、相互依存の条件は次のようになる。

1. $g_i \in T_{r/l} \wedge g_i \in Prec(g_s)$ であるゴール $g_i, T_{r/l}$ $(r/l \neq q/m)$ が存在し、

2. $g_j \in T_{r/l} \wedge g_s \in Prec(g_j)$ であるゴール g_j が存在する。

ループ p/n についてこの条件が満たされるとき、すべての再帰ゴール p/n でスレッド切替チェックを行う必要がある。そこで、各再帰ゴール p/n に、本研究で新設したプラグマ@loop を付加する。

図 5.10の例では、ループ p/n は他のスレッドと相互依存を持つので、すべての再帰ゴール p/n に@loop プラグマを付加し、再帰ごとにスレッド切替チェックを行う。しかし、ループ q/m は単方向の依存しか持たないため、@loop プラグマは付加しない。

5.4.7 reply first 方式のための解析

5.3.2 項で述べたように、reply first 方式を実現するには、個々の変数を具体化するスレッドを解析する必要がある。また、スレッド間で共有されない変数は reply first 方式の対象にならないから、共有の有無を区別する必要がある。

図 5.18, 5.19 の型解析アルゴリズムにより、互いに同一化する可能性のある変数の集合 $U_1, \ldots, U_{umax}$ が得られる。これを利用して、共有変数を次のように判定できる。

スレッド p/m に含まれるゴールの集合を $T_{p/m}$ としたとき、ある $v_i, v_j \in U_k$ $(1 \leq k \leq umax)$ について、$g_k(\ldots, v_i, \ldots) \in T_{p/n} \wedge g_l(\ldots, v_j, \ldots) \in T_{q/m}$ $(p/n \neq q/m)$ が成り立つならば、U_k に含まれる変数はすべて共有変数である。

共有変数を具体化するスレッドは型集合から得ることができるが、実行時に同じスレッドの実体が複数生成される場合や、節選択の非決定性により具体化スレッドが一意に決定しない場合がある。そこで、本研究では静的に具体化スレッドを解析せず、共有変数に自身を具体化するスレッドへのポインタを保持させるコードを生成し、実行時にこれを適時更新していく方式をとる。これを実現するために必要な解析情報は、前記の型解析アルゴリズムより得られた、スレッド開始ゴールの入出力モードである。

5.4.8 解析例

以上で述べてきた解析手法の適用例として、図 5.2のプログラム handshake2 を取り上げる。

型解析 5.4.3 項に示した型解析を行うと、図 5.25のような結果が得られる。このプログラムはループ p0/2 および p1/2 を含んでいるが、5.4.3 項の拡張により見掛けの依存が解消され、$\{(1, \{2\})\}$ のように節 1 のゴール p0/2, p1/2 間の依存に置き換えられている。

ゴール間の依存解析 図 5.26のようなゴール間の順序制約が得られる。依存ゴール集合中の $\{(1, \{2\})\}, \{(2, \{1\})\}$ がここで展開され、節 1 のゴール p0/2, p1/2 間の相互依存を生じている。

プログラムのスレッド化 節 1 はゴール間に相互依存があるため、p1/2 に@thread を付加する。これによって、このプログラムは節 1,2,3 から成るスレッド main/0 と、スレッド main/0 より生成されるスレッド p1/2 に分割される。

ループ構造の解析 ループ p0/2, p1/2 についてループ内外の相互依存解析を行う。スレッド main/0 内のループ p0/2 について、ループ開始ゴールは $p0/2^1$ であり、図 5.26より $2 \in Prec(1)$ であり、ゴール $p0/2^2$ はスレッド p1/2 内である。また、$Prec(2)$ に 1 が含まれている。よって、このループはループ外との相互依存を持つ。ループ p1/2 についても同様にして、相互依存を持つことが判る。したがって、再帰ゴール $p0/2^6$ および $p1/2^{13}$ に、プラグマ@loop を付加する。

```
1 main :- p0[1]([int(100)|L1],L2), p1[2](L1,L2).
2 p0(L1,L2) :- L1=[3][int(N)|NL1]
             | L2=[4][int(NN)|NL2], NN:=[5]N-1, p0[6](NL1,NL2).
3 p0(L1,L2) :- L1=[7][] | L2=[8][].
4 p1(L1,L2) :- L2=[9][int(N)|NL2], N>[10]0
             | L1=[11][int(NN)|NL1], NN:=[12]N-1, p1[13](NL1,NL2).
5 p1(L1,L2) :- L2=[14][int(N)|NL2], N=:=[15]0 | L1=[16][].
```

$$
\begin{aligned}
T(\mathtt{L1}^1) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^4,\{2\})\})),\{2\})\},\{(NL1^4,\{2\})\})),\{2\}),\\
&\quad (a,\{2\})\}\\
T(\mathtt{L2}^1) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^2,\{1\})\})),\{1\})\},\{(NL2^2,\{1\})\})),\{1\}),\\
&\quad (a,\{1\})\}\\
T(\mathtt{L1}^{2,3}) &= \{\ (s(.,2,(\{(s(\mathtt{int},1,(\{(i,\{1\}),(NN^4,\{(1,\{2\})\})\})),\{(1,\{2\})\})\},\\
&\qquad\qquad \{(L1^1,\{1\}),(NL1^4,\{(1,\{2\})\})\})),\\
&\qquad \{(1,\{2\})\})\\
&\quad (a,\{(1,\{2\})\})\}\\
T(\mathtt{L2}^2) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^2,\{4\})\})),\{4\})\},\{(NL2^2,\{4\})\})),\{4\})\}\\
T(\mathtt{NL1}^2) &= \{(L1^1,\{3\}),(NL1^4,\{(1,\{2\}),3\})\}\\
T(\mathtt{NL2}^2) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^2,\{6\})\})),\{6\})\},\{(NL2^2,\{6\})\})),\{6\}),\\
&\quad (a,\{6\})\}\\
T(\mathtt{N}^2) &= \{(i,\{1\}),(NN^4,\{(1,\{2\}),3\})\})\}\\
T(\mathtt{NN}^2) &= \{(i,\{5\})\}\\
T(\mathtt{L2}^3) &= \{(a,\{8\})\}\\
T(\mathtt{L1}^4) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^4,\{11\})\})),\{11\})\},\{(NL1^4,\{11\})\})),\{11\})\}\\
T(\mathtt{L2}^{4,5}) &= \{\ (s(.,2,(\{(s(\mathtt{int},1,(\{(NN^2,\{(2,\{1\})\})\})),\{(2,\{1\})\})\},\\
&\qquad\qquad \{(NL2^2,\{(2,\{1\})\})\})),\\
&\qquad \{(2,\{1\})\}),\\
&\quad (a,\{(2,\{1\})\})\}\\
T(\mathtt{NL1}^4) &= \{(s(.,2,(\{(s(\mathtt{int},1,(\{(NN^4,\{13\})\})),\{13\})\},\{(NL1^4,\{13\})\})),\{13\}),\\
&\quad (a,\{13\})\}\\
T(\mathtt{NL2}^4) &= \{(NL2^2,\{(2,\{1\}),9\})\}\\
T(\mathtt{N}^4) &= \{(NN^2,\{(2,\{1\}),9\})\})\}\\
T(\mathtt{NN}^4) &= \{(i,\{12\})\}\\
T(\mathtt{L1}^5) &= \{(a,\{16\})\}\\
T(\mathtt{NL2}^5) &= \{(NL2^2,\{(2,\{1\}),14\})\}\\
T(\mathtt{N}^5) &= \{(NN^2,\{(2,\{1\}),14\})\})\}
\end{aligned}
$$

$m(\mathtt{p0}/2,1) = i,\ m(\mathtt{p0}/2,2) = o,\ m(\mathtt{p1}/2,1) = o,\ m(\mathtt{p1}/2,2) = i$

図 5.25: handshake の型解析結果

$$
\begin{aligned}
Prec(1) &= \{2\} \\
Prec(2) &= \{1\} \\
Prec(5) &= \{3\} \\
Prec(10) &= \{9\} \\
Prec(12) &= \{9\} \\
Prec(15) &= \{14\} \\
\text{上記以外の } Prec(i) &= \emptyset
\end{aligned}
$$

$$\Downarrow$$

ゴール間の順序制約は
$prec(1,2), prec(2,1)$

図 5.26: handshake のゴール間依存解析結果

$$
\begin{aligned}
U_1 &= \{\mathtt{L1}^1, \mathtt{L1}^2, \mathtt{NL1}^2, \mathtt{L1}^3\mathtt{L1}^4, \mathtt{NL1}^4, \mathtt{L1}^5\} \\
U_2 &= \{\mathtt{L2}^1, \mathtt{L2}^2, \mathtt{NL2}^2, \mathtt{L2}^3\mathtt{L2}^4, \mathtt{NL2}^4, \mathtt{L2}^5, \mathtt{NL2}^5\} \\
U_3 &= \{\mathtt{N}^2, \mathtt{NN}^4\} \\
U_4 &= \{\mathtt{NN}^2, \mathtt{N}^4, \mathtt{N}^5\}
\end{aligned}
$$

図 5.27: handshake の同一化集合

```
1 main :- p0[1]([int(100)|L1],L2), p1[2](L1,L2)@thread.
2 p0(L1,L2) :- L1=[3][int(N)|NLi]
             | L2=[4][int(NN)|NL2], NN:=[5]N-1, p0[6](NL1,NL2)@loop.
3 p0(L1,L2) :- L1=[7][] | L2=[8][].
4 p1(L1,L2) :- L2=[9][int(N)|NL2], N>[10]0
             | L1=[11][int(NN)|NL1], NN:=[12]N-1, p1[13](NL1,NL2)@loop.
5 p1(L1,L2) :- L2=[14][int(N)|NL2], N=:=[15]0 | L1=[16][].
```

図 5.28: handshake のスレッド化

共有変数の判別 図 5.27の同一化集合より、プログラム中の全変数はU_1, U_2, U_3, U_4のいずれかにふくまれ、かつ各集合はスレッド main/0, p1/2 の双方の変数を含んでいる。したがって、このプログラムでは全変数が共有変数である。

以上の解析の結果、スレッド化されたプログラムは図 5.28のようになる。

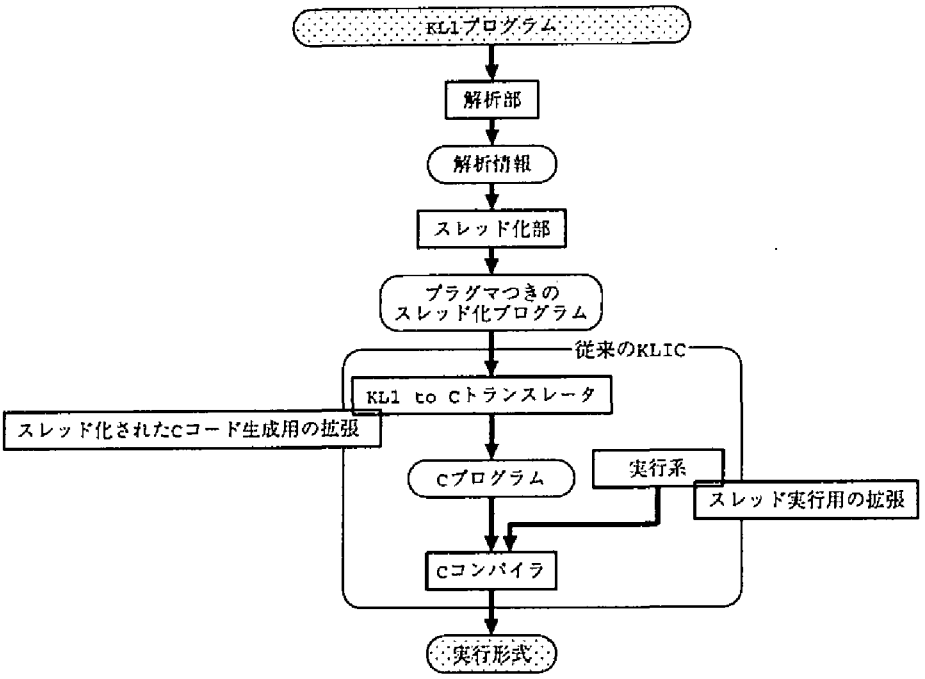


図 5.29: ゴール・スケジューリング最適化版 KL1 処理系

5.5 実装

前節での静的解析結果とスレッド化されたプログラムを基に、本節では KL1 処理系 KLIC を対象にした実装について述べる。なお、本節の内容のうち、トランスレータおよび実行系の設計・実装は、伊川雅彦氏によるものである [10, 56]。

5.5.1 拡張処理系の構成

本研究では、KLIC に対し拡張・変更を加える形で、処理系の実装を行った。処理系全体の構成を、図 5.29に示す。以下、本節では実装した処理系を拡張版処理系、元の KLIC 処理系を従来版処理系と呼ぶ。

拡張版処理系では、入力された KL1 プログラムに対し、まず 5.4 節で述べた静的解析を行う。続いて、スレッド化を行い、プラグマやゴール引数のモード情報などを付加したプログラムを KL1 to C トランスレータに引き渡す。

トランスレータでは 2.3.2 項で述べたように、ヘッドとのパターンマッチングやガード部の検査、ゴールレコードの生成などを行う C コードを生成する。そこで、この部分でスレッド実行を行う C コードを生成するよう、トランスレータを拡張・変更した。

また、スレッド実行用のスタック管理や reply first スケジューリングを実現するノード間通信などは、実行系によるサポートが必要である。そこで、これらの機能を実現するよう、従来の実行系に拡張・変更を加えた。

以下、拡張版のデータ管理方法について述べた後、トランスレータ部と実行系の拡張について説明する。

5.5.2 データの管理方法

スレッドとゴール

従来版ではゴールをヒープ上のゴールレコードで表現し、このレコードをゴールレコードキューにつなぐことで、実行可能なゴールを管理する。また、中断したゴールはそのゴールレコードを中断原因となった変数にフックする。これに対し、拡張版ではスレッドを、ヒープ上のスレッドレコードと個々のスレッドごとに割り当てられたゴールスタックで表現し、ゴールレコードはこのゴールスタック上に生成する。

スレッドレコード　動的スケジューリングの単位として、従来のゴールレコードと同様にヒープ上に生成し、スレッドレコードキューによって実行可能スレッドを管理する。1個のレコードは、自身のゴールスタックへのポインタの他、要求レコードのFIFOキューを保持する。後者はreply firstスケジューリングを行う際に、そのスレッドに対する変数の具体化要求を管理するためのものである。

ゴールスタック　ヒープ領域とは別に一定個数が静的に確保され、スレッドの生成ごとに空いているものが順に割り当てられる。スタックの大きさが足りなくなった場合には、新たなスタックを割り当て、スタックのチェーンを形成する。また、処理系全体でスタックの個数が足りなくなった場合には、動的に再確保が行われる。

要求レコード　reply firstスケジューリングを行う場合には、ノード間通信によるreadメッセージ受信や実行中スレッドの中断が生じると、実行系は必要な変数を具体化するスレッドをスケジューリングする。要求レコードは、このスレッド切替を生じた未具体化変数へのポインタを保持し、複数の具体化要求が生じた場合などの順序管理に使用される。

ゴールレコード　従来版と同様に、対応する実行コードへのポインタ、および引数の並びから成る。ただし、ヒープではなくゴールスタック上に生成されるため、キューを形成するためのポインタは持たない。

共有変数

reply first方式を実現するためには、共有変数について具体化スレッドを知る必要がある。5.4.1項で述べたように、本手法ではこれを実現するために、共有変数には具体化スレッドへのポインタを保持させ、実行時にこの内容を適時更新していくという方式を取る。

2.3.1項で述べたように、従来版の実装では、変数はREFタグつきのセルで表され、格納されたポインタの値が自分自身のアドレスと一致するかどうかで未具体化かどうか判断される。したがって、そのままでは未具体化変数にスレッドへのポインタを保持させることができず、新たに共有変数用のデータ構造を作る必要がある。

ジェネリック・オブジェクトとして共有変数を実現すれば、処理系核の同一化処理などを変更する必要がなく、実装が用意である。しかし、共有変数の生成や操作は頻繁に生じるため、これらすべてをメソッド呼出によって行うのは、非常に大きなオーバヘッドとなる。そこで、本研究の実装では、表2.2のREFタグ付セルに共有／非共有の区別を行うためのタグを1ビット追加し、このビットが1の場合は共有変数であるとする。

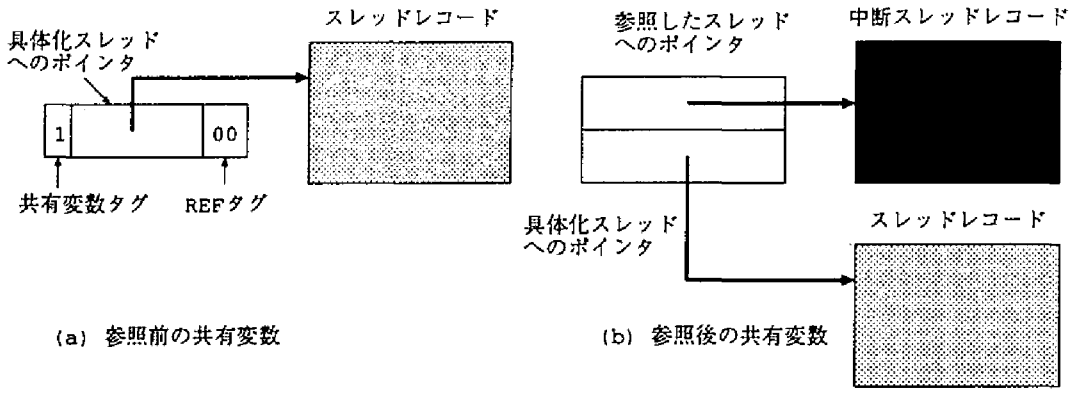


図 5.30: 共有変数の実装

共有変数には、生成された時点で、自身を具体化するスレッドのスレッドレコードへのポインタが格納される (図 5.30(a))。具体化スレッドは実行時に動的に追跡されるため、このポインタの値は 5.5.4 項で述べるように実行の過程で適時変更される。また、この共有変数が具体化される前に参照されると、具体化スレッドおよび参照を行ったスレッドのスレッドレコードを指すデータ構造に変化する (図 5.30(b))。

5.5.3 トランスレータの拡張

解析部・スレッド化部の出力は、もとの KL1 プログラムに対し、本研究で拡張したプラグマ @thread および@loop が追加されている。また、各ユーザ定義述語の引数のモード情報と、変数の共有／非共有情報が付加されている。

拡張版トランスレータはこれを入力とし、スレッド単位のゴール実行を行うコードを出力する。従来版との変更点を以下に説明する。

ゴールの生成

ガード部の検査が成功し、ある節が選択されると、C コードでは従来版と同様に、その節のボディゴールに対応するゴールレコードを生成し、引数を設定する。拡張版でもゴールレコードの形式は変わらないが、従来版のようにヒープ上に生成する代わりに、現在のスレッドに割り当てられているゴールスタック上に生成する。また、従来版では生成したゴールレコードを実行可能ゴールキューにつなぐが、ゴールスタック上のゴールレコードは LIFO で実行されるため、ポインタのリンクによるキューの形成は不要である。したがって、拡張版ではこの操作を行うコードは生成しない。

プラグマの処理

各種のプラグマが付加されたゴールについては、それぞれ次のような処理を行うコードを生成する。

goal@thread　新スレッドの開始を指示するプラグマであるので、新たなスレッド環境を生成し、*goal* から次の@thread つきゴールの一つ前までのゴールレコードを、新スレッドのゴールスタッ

```
p :- q, r, s@thread, t, u@thread.
```

図 5.31: @thread によるスレッド生成

ク上に生成する。

たとえば、図 5.31の節に対しては、次のような処理を行うコードを生成する。

1. 新スレッド u の環境を生成し、そのゴールスタック上に u のゴールレコードを積む。
2. 新スレッド s の環境を生成し、そのゴールスタック上に t, s のゴールレコードを順に積む。
3. 現在のスレッドのゴールスタック上に、r, q のゴールレコードを順に積み、q に対応するコードに分岐する。

goal@loop　*goal*は、ループ外との相互依存性を持つループの再帰ゴールである。そこで、まず現在のゴールスタックに *goal* を積み、続けてスレッド切替のチェックを行う疑似ゴールを積む。実行時には、まず疑似ゴールが実行されて実行再開したスレッドの有無をチェックし、もし存在すればそちらに実行を切り替える。該当するスレッドがなければ、このまま現在のスレッドの実行を続けるので、次に再帰ゴール *goal* が実行される。この結果、ループ末での resumption first 方式のスレッド切替が実現される。

goal@node(N)　並列実行時に他ノード Nへのゴール送信を指示するプラグマである。このプラグマ自体は従来版から存在しているが、スレッド実行に対応するため、拡張版では生成するコードに変更を加える。

ノード Nに送信されたゴールは、そこで新たなスレッドを生成し、その開始ゴールとなる。そこで拡張版のコードでは、まず新たなスレッド環境を生成し、そのゴールスタックに *goal* のゴールレコードを積む。続いて、このスレッド環境をノード Nに送信する。

また、5.5.4 項で述べる実行系の拡張により、ノード Nでは受信したスレッド環境より、自ノード内に新スレッドを生成する。

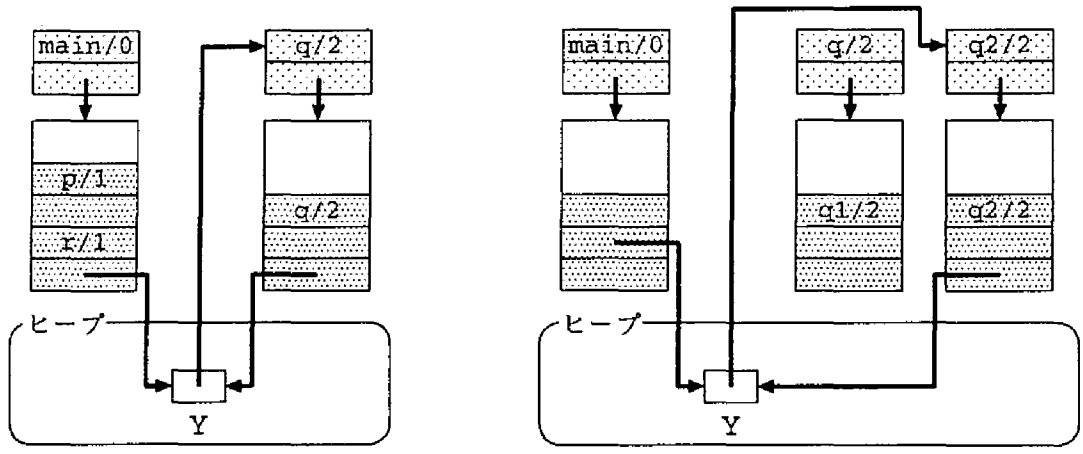
共有変数の操作

非共有変数の操作については従来と同じコードを生成するが、未具体化の共有変数については、実行時に具体化スレッドの追跡を行う必要がある。そこで、拡張版では生成する C コードに以下の変更を加える。

1. 節 c において、共有変数 X がボディ部のみに出現するなら、この節に対応するコード中でヒープ上に X を生成する。また、c のボディゴールのうちで、X を出力引数に取るゴール g_i が、必ず 1 個存在する。そこで、g_iを含むスレッドのレコードへのポインタを X に保持させる。
2. 節 c において、クローズヘッドの引数に共有変数 X が出現する場合、X はこの節の呼出以前にヒープ上に生成されている。このとき、c のボディゴール g_iの出力引数に X が出現し、g_i が c で新たに生成されるスレッド T_jに含まれる場合、X の保持する具体化スレッドのレコードへのポインタを、T_jのレコードへのポインタに書き換える。

```
1 main :- p(X^o), r(Y^i), q(X^i,Y^o)@thread.
2 q(X^i,Y^o) :- q1(X^i,Z^o), q2(Z^i,Y^o)@thread.
```

※ X^mは、その変数引数のモードがmであることを表す



(a) 節1実行後　　(b) 節2実行後

図 5.32: 共有変数の具体化スレッド追跡

例として、図 5.32のプログラム片を考える。節 1 では共有変数 Y が生成されるが、これを出力引数に持つゴール q/2 は新スレッドを生成するので、Y にはスレッド q/2 へのポインタが格納される (図 5.32(a))。次に、節 2 では新たにスレッド q2/2 が生成されるが、これは Y を出力引数に持つ。したがって、節 2 を実行した時点で、Y はスレッド q2/2 へのポインタに書き換えられる (図 5.32(b))。

5.5.4 実行系の拡張

実行系は従来版のものを基に、以下の変更・拡張を行った。

スレッドの実行

従来版・拡張版ともに、ゴールレコードの生成はトランスレータが生成する C コード内で行われるが、次の実行ゴールレコードの取り出しは実行系のスケジューラにより行われる。従来版ではこの部分の処理は、ゴールレコードキューの先頭レコードを取り出す形で行われるが、拡張版ではゴールレコードがゴールスタックに保持されているので、単にスタックトップのゴールレコードが指すコードに分岐する。

ノード間通信

ノード間のメッセージ通信のうち throw メッセージには、従来版のゴールの代わりに、送信ゴールを開始ゴールとするスレッドの環境が格納されている。そこで、このメッセージ受信時には、受け取ったスレッドを新たに自ノード上に生成する。また、read メッセージを受信したときには、次項に述べる reply first スケジューリングを行う。

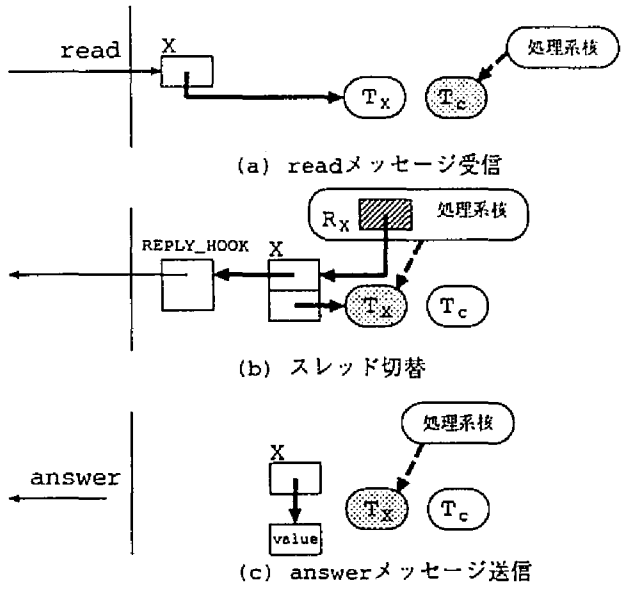


図 5.33: reply first スケジューリング

スレッドの reply first スケジューリング

他ノードからの read メッセージを受信したときや、実行中のスレッドが中断したときは、実行系が reply first スケジューリングを行う。以下、具体的な方法を説明する。

具体化スレッドのスケジューリング ノード N_cが変数 X を参照し、ノード N_pに read メッセージが送信される場合を考える。N_pで実行中のスレッドを T_c、X を具体化するスレッドを T_Xとすると、次のようなスケジューリングが行われる。

1. read メッセージ受信時 (図 5.33(a)) に、要求された変数 X が未具体化の共有変数であった場合は、X は図 5.30(b) の形に変化し、返信用の REPLY_HOOK と T_Xを指す。また、処理系核には、reply first 方式で具体化すべき変数として、X へのポインタを持った要求レコード R_Xが与えられる。

2. T_cの実行中のゴールリダクションが終了した時点で、要求レコードを保持しているため、これが指す共有変数 X の具体化スレッド T_Xに実行を切り替える。(図 5.33(b))。

3. T_X中のあるゴールにより X が具体化された時点で、REPLY_HOOK の unify メソッドが呼び出され、answer メッセージによる返信が行われる (図 5.33(c))。

以上の動作により、read メッセージ到着後、answer メッセージが返信されるまでの時間が最小化される。このため、ノード N_c側が他に実行可能スレッドを持たない場合、アイドルとなる時間を最小限にすることができる。

具体化スレッドが中断する場合 上記の例で、X の具体化スレッド T_Xが、変数 Y への参照で中断する場合を考える。この場合、Y を具体化するのは他のスレッド内のゴールであるから、Y も共有

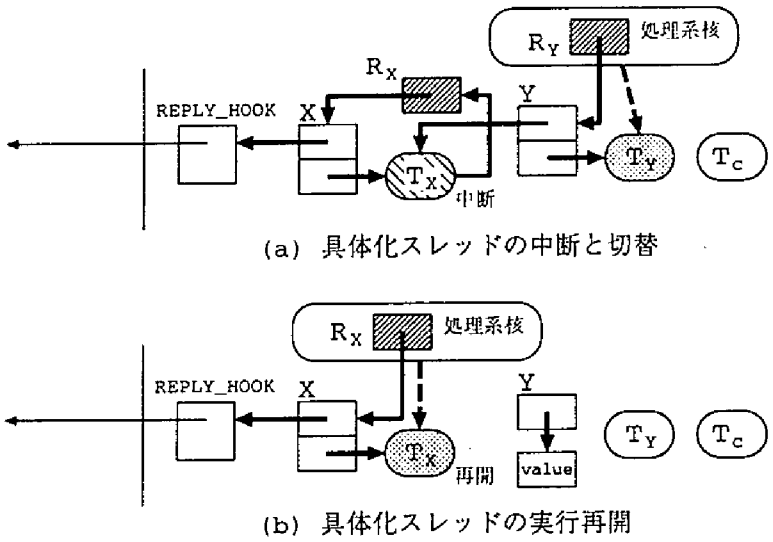


図 5.34: 具体化スレッドの中断

変数であり、自身を具体化するスレッド T_Yへのポインタを持つ。このとき、次のようにして、中断した T_Xの要求する Y についての reply first スケジューリングが行われる。

1. T_Xは X の要求レコード R_Xにフックして中断し、処理系核には新たな要求レコード R_Yが与えられる。この結果、R_Yが指す変数 Y の具体化スレッド T_Yがスケジューリングされる (図 5.34(a))。

2. T_Yが Y を具体化すると、T_Xが実行再開する。この時点で、スレッド T_Xに実行が切り替わる (図 5.34(b))。

この動作によって、read メッセージにより要求された値を生成するまでに他のスレッドの動作が必要な場合は、次々に要求駆動的なスケジューリングが行われる。また、具体化スレッドの中断原因が解消された時点で元のスレッドに実行が戻されるため、最初に値を要求された変数 X に関係がある部分のみが実行される。この結果、T_Xの中断が起きる場合でも、やはり最小限の時間で X が具体化される。

具体化ゴールを含むスレッドが未生成の場合 X の具体化ゴールを含むスレッド T_Xが生成される前に、X の read メッセージが到着する可能性がある。しかし、5.5.3 項で述べたように、共有変数 X には各リダクションの時点で、X を出力引数とするゴールを含むスレッドへのポインタが格納される。そして、このようなスレッドは、X の具体化ゴールを含むスレッドか、その先祖スレッドのどちらかである。したがって、X のポインタが T_Xを指していない場合にも、現在ポインタが指している T'_Xを実行する。また、この結果、新たにスレッド T''_Xが生成された場合は、T''_Xが X を出力引数とするゴールを含むなら T''_Xに実行を切り替え、含まないならそのまま T'_Xの実行を続ける。この動作によって、最短時間で T_Xが生成される。

複数の具体化要求が存在する場合 変数 X への read メッセージが到着し、これに対する reply first スケジューリングを行っている最中に、別の変数 Z への read メッセージが到着する可能性が

ある。この場合は、その時点で要求レコード R_Z を生成し、処理系核の要求レコードキューに追加する。そして、変数 X の返信が終了し、処理系核が R_X を解放した時点で、要求レコードキューから処理待ちの要求レコード R_Z を取り出し、同様に reply first スケジューリングを行う。

5.6 性能評価

本節では、前節の実装によるゴール・スケジューリング最適化手法の性能評価について述べる。

5.6.1 評価環境

評価用の環境として、KLIC version 2.002 に前節の実装を行い、イーサネットで結合した 1〜4 台のワークステーション (SS20) で評価を行った。また、実行時の使用メモリは、ヒープ領域を 1 メガワード、ゴールスタック領域を全体で 1 キロワード使用した。並列実行の場合には、KLIC の並列実行管理機能に 1 プロセッサを割り当てている。

なお、現在の版では解析部に共有変数の判定が実装されていないため、reply first 方式を用いる場合には、全変数を共有変数として扱っている。

5.6.2 評価プログラム

評価プログラムには、次の 5 本を用いた。

`handshake2` 図 5.2に挙げた、2 個の並列プロセスが 2 本のストリームで整数値をやり取りするプログラムである。ただし、5.2.1 項で述べた問題を生じるように、ゴール 4,5,11,12 をユーザ定義述語に書き換えてある。

`reverse` あらかじめ与えられた着手の系列にしたがって、2 個の並行プロセスがオセロゲームを行うプログラムである。

`circle_to_square` 円内の格子点の数を数え上げることで、直径 d の円と一辺 d の正方形の面積比を求めるプログラムである。

`master_slave` 複数のスレーブプロセスがそれぞれ整数のリストを生成し、マスタープロセスがそれらの値を使用するプログラムである。

`nqueen` 4.7 節で用いたのと同じ、N-queen 問題を解くプログラムである。問題のサイズは $N = 12$ としている。

5.6.3 実行結果

`handshake` 実行結果を表 5.1に示す。このプログラムには並列性がほとんどないため、逐次実行のみ評価を行った。また、全体でスレッドが 2 本しか生成されないため reply first 方式は意味がないが、オーバヘッド量を評価するため、この方式を用いたときの結果も挙げてある。

スレッド化により中断回数は 1/2 になっている。これは、5.2.1 項で述べたように、構造型データの不完全な具体化により並行プロセスの切替回数が増加するという、resumption first 方式の問題点が解消されたためである。また、ゴールレコードをスタック上に生成することにより、ヒープの消費量が大幅に減少し、GC 回数が約 1/3 になっている。これらの効果に加え、ゴールレコードキューの管理オーバヘッドがなくなったことにより、全体では約 2 倍の速度向上が得られている。

表 5.1: handshake2 の実行結果

	実行時間 (秒)	GC 回数	中断回数
従来版	19.21	24	1000000
スレッド化	9.42	7	500001
reply first	10.77	11	500001

表 5.2: reverse の実行結果

	実行時間 (秒)	GC 回数	中断回数
従来版	79.42	119	2964000
スレッド化	27.37	17	92999

また、reply first 方式を用いた場合は、単なるスレッド化に比べ、1 割程度の速度低下が起きている。これは、共有変数操作と reply first 方式によるスレッド切替のオーバヘッドによるものである。また、参照後の共有変数が 2 セル分のデータ構造に変化するため、ヒープを余分に消費して GC 回数を増加させることも、速度低下の原因になっている。

reverse 実行結果を表 5.2に示す。このプログラムも並列性がなく、スレッドが 2 本しか生成されないため、逐次環境での従来版とスレッド化の結果のみ挙げてある。

このプログラムでは、ゲームの着手はあらかじめ与えられているため、処理内容は並行プロセス間で交互にゲームの盤面を更新する (着手にしたがって石を引っくり返す) だけである。盤面はファンクタとリストを用いて表現されており、従来版では盤面の一部が具体化されるごとに、resumption first 方式によって相手プロセスに制御が移ってしまう。このため、非常にたくさんの無駄なプロセス切替が生じる。

一方、スレッド化によってプロセス切替は最小限になるため、中断回数は約 1/32 と激減している。また、この問題のように多重ループで多次元の構造型データの各要素を参照／変更するプログラムは、ループの 1 要素ごとがゴールとなるため、大量のゴールレコードを生成する。スレッド化によって、これらのレコードがヒープを消費しなくなるため、GC 回数は約 1/7 と大きく減少している。

以上の効果によって、スレッド化することにより、全体では約 2.9 倍の速度向上が得られている。

circle_to_square 実行結果を表 5.3に示す。このプログラムは、それぞれ異なる d の値について 2 ノードで並列に計算を行う。したがって、逐次環境、および 2 台 (+荘園 1 台) での結果を挙げてある。

逐次実行では、従来版でも中断はほとんど生じないため、スレッド化を行っても中断回数削減の効果はない。しかし、スタックの利用により、GC 回数とゴールレコードキューの管理オーバヘッドの削減効果は得られるので、約 1.3 倍の速度向上を達成している。

並列実行では、2 ノードでの計算はそれぞれ独立して行えるため、従来版で逐次実行に対し約 1.6 倍の台数効果を得ている。しかし、これを単純にスレッド化すると、GC 回数は削減されているにも関わらず、実行時間は逐次のスレッド化と同程度であり、従来版の並列実行に対し、か

表 5.3: circle_to_square の実行結果

	実行時間 (秒)	GC 回数	中断回数	アイドル時間 (秒)	
				ノード 0	ノード 1
逐次					
従来版	53.03	84	3	–	–
スレッド化	39.61	30	1	–	–
並列 (2+1) 台					
従来版	33.00	84	9	10.72	0.06
スレッド化	40.94	30	2	24.40	16.56
reply first	25.43	30	2	8.34	0.13

えって遅くなっている。これは、5.3.2 項で述べた、スレッドの並列実行によるアイドル時間の増加 (図 5.8) という問題が生じているためである。

このプログラムでは、短時間で計算の初期値を生成するゴール群と、ノード 0,1 で長時間にわたり面積比の計算を行うゴール群とが、各々別スレッド (それぞれ T_I, T_{C0}, T_{C1} とする) になる。そして、ノード 0 で T_I, T_{C0}、ノード 1 で T_{C1} が、それぞれ生成される。ノード 0 では最初 T_{C0} がスケジューリングされ、初期値を参照した時点で中断して T_I に切り替わり、T_{C0} が参照する値が生成された時点で、再帰呼出前の resumption first 方式により再び T_{C0} に切り替わる。ところが、T_{C0} が再開した時点での T_I の未実行部分は、T_{C1} の参照する初期値を計算するゴール群である。このため、単純なスレッド化では、T_{C0} が実行を終了して T_I の残りの部分が実行されるまで T_{C1} は中断したままになり、他にスレッドを持たないノード 1 はその間アイドルになる。このため、ノード 0 の処理がほとんど終了するまで、ノード 1 の処理がまったく進まず、逐次実行と変わらなくなってしまう。

そこで、reply first スケジューリングを行うと、ノード 1 が初期値を参照し、ノード 0 に read メッセージを送信した時点で、参照変数を具体化するスレッド T_I がスケジューリングされ、ノード 1 への値返信が行われるまで、T_{C0} の実行は後回しにされる。このため、T_{C0} と T_{C1} が並列に実行され、逐次のスレッド化に比べて約 1.6 倍、従来版の並列実行と比べても約 1.3 倍の速度向上が得られている。

master_slave　逐次環境、および 2 台 (+荘園 1 台) での実行結果を、表 5.4に示す。

このプログラムは、複数のスレーブプロセスがそれぞれ整数リストを生成し、それらの要素をマスタープロセスが集計する。マスターおよびスレーブのループは、組込述語と再帰ゴールのみの非常に簡単な形をしている。このため、KLIC では最適化によりゴールレコードをあまり作らないコードが生成される。この結果、ゴールの中断の起きない逐次実行では、スレッド化による効果はまったく得られない。

一方、並列実行では、circle_to_square と同様にアイドル時間の問題が発生する。このプログラムは、複数のスレーブプロセス $P_{S1}, \ldots, P_{Sn}$ がそれぞれ整数リスト $L_1, \ldots, L_n$ を出力するのに対し、マスタープロセス P_M は L_1 の先頭要素, $\ldots$, L_n の先頭要素, L_1 の第 2 要素, $\ldots$ のように、順に 1 要素ずつ参照していく。このため、ある P_{Si} だけが実行されても、次の P_{Si+1} が実行されるまで、P_M は処理を進めることができない。ところが、ノード 0 に P_M、ノード 1 に $P_{S1}, \ldots, P_{Sn}$

表 5.4: master_slave の実行結果

	実行時間 (秒)	GC 回数	中断回数	アイドル時間 (秒)	
				ノード 0	ノード 1
逐次					
従来版	60.40	0	0	–	–
スレッド化	60.42	0	0	–	–
並列 (2+1) 台					
従来版	70.78	0	3108	39.49	31.00
スレッド化	69.44	0	1019	37.86	34.93
reply first	44.07	0	1020	6.59	9.27

表 5.5: nqueen の実行結果

	実行時間 (秒)	GC 回数	中断回数
逐次			
従来版	40.80	68	0
スレッド化	29.79	21	0
reply first	36.73	35	0
並列 (2+1) 台			
従来版	23.27	66	221371
スレッド化	17.31	20	57
reply first	20.40	36	54
並列 (3+1) 台			
従来版	16.20	66	338687
スレッド化	12.27	21	87
reply first	13.71	36	78

のように割り当てると、ノード 1 の各スレーブプロセスは互いに依存がないため、P_{Si}が最後まで実行されてから P_{Si+1}が実行を開始する。この現象は従来版・スレッド化版の両方で発生し、通信オーバヘッドも含めると逐次実行よりかえって遅くなってしまう。これに対し、reply first 方式では、P_Mが参照した値を生成するスレッドが次々にスケジューリングされる。このため、従来版に比べてアイドル時間が 1/4 程度に減少し、速度が約 1.6 倍に向上している。また、逐次の従来版と比べても、約 1.4 倍の台数効果を得ている。

nqueen 逐次環境、および 2,3 台 (+荘園 1 台) での実行結果を表 5.5に示す。

このプログラムは粗粒度の負荷分散を行うため、従来版でも台数効果は得られているが、スレッド化により台数効果を保ったまま、1.3〜1.4 倍程度の速度向上を得ることができる。また、アイドル時間の問題が生じないプログラムであるため、reply first 方式を用いると、オーバヘッドのためかえって実行速度が低下する。しかし、この場合でも従来版に比べ 1.1〜1.2 倍程度の速度向

上を達成している。

5.7 関連研究

本節では、本章のゴール・スケジューリング最適化手法に関連する研究について述べる。

5.7.1 Massey らの研究

Massey らの研究 [57] では、Flat GHC の逐次化手法を提案している。

この手法では、対象プログラムについて、ゴールの引数のモードがすべて決定可能 (fully-moded) であり、同じボディゴールの引数の間にサイクリックな依存関係が存在しない (feedback-free) ことを前提とし、モード解析結果を用いて逐次化を行う。しかしながら、既存の多くのプログラムはこの前提条件を満たしていない、という問題がある。これに対して、本研究の手法は対象プログラムが well-moded であることしか前提としておらず、より多くのプログラムを扱うことができる。

5.7.2 荒木らの研究

荒木らの研究 [54] は、commited-choice 型の並列論理型言語 Fleng [58] を対象に、ゴールの粒度制御手法を提案している。

この手法では、ゴール g_iの呼出を g_iの定義節のゴール群で置き換えるインライン展開と、複数のゴールの定義節を一つの節にまとめ、これらのゴールを一つにするゴール融合という、2 種類の方法を用いている。5.3.1 項で述べたように、この方法はゴールそのものの数を減らすためゴール環境の生成・管理オーバヘッドを減少させるという利点があるが、共有述語をインライン展開する場合はコード量が増加するという問題がある。また、スケジューリングの単位はあくまでゴールであるので、本研究のスレッドのように、スケジューリング単位の中にループ全体を含めることはできない。また、この手法ではプログラム自体を変形するため動的な粒度変更ができないが、バンバン粒度制御 [59] を利用し、高負荷・低負荷用の 2 種類のコードを動的に切り替えることで問題を解決している。

依存解析についてはモード解析を利用しているが、ゴールの呼出木上でボトムアップの情報伝達しか行っておらず、被呼出側の解析に呼出側の情報を利用できない。また、構造型データについては引数に関する依存を解析せず、そうした依存は可能性があるものすべてを存在するとして処理している。本研究の手法はこれらの解析も行っているため、より正確な依存関係を得ることができ、結果的により高い粒度が達成できる。

5.8 結言

本章では、並列論理型言語 KL1 のゴール・スケジューリング最適化手法について述べた。

KL1 などの並行／並列論理型言語では、並行制御の単位となるゴールが細粒度であるため、スケジューリングのオーバヘッドが非常に大きなものになる。本研究では、静的解析情報を用いてゴール間の依存解析を行い、実行順序を固定可能なゴール群をスレッドとして逐次化することで、並行実行の粒度を大きくし、スケジューリングオーバヘッドを削減する手法を提案した。

本手法では、型解析をベースとした依存解析手法により、構造型データの引数を介した依存などを含む、詳細な解析を行っている。また、実行順序のみ固定し、ゴールの枠組みは残している

ため、1スレッド内にループ構造を含めることができる。これらのことから、非常に大きなスレッドを生成でき、動的スケジューリングの回数を大幅に減らすことができる。さらに、スレッド内のゴール環境をスタックで保持することにより、ヒープの消費量が減少し、ガーベジコレクションのオーバヘッドも削減できる。

また、スレッド化によって並行実行の粒度を大きくしたことにより、ノード間の依存によるアイドル時間が増加する問題が生じる。この解決法として、要求データを生成するスレッドを優先的にスケジューリングする reply first 方式を提案した。

性能評価の結果、本手法によりゴール中断やガーベジコレクションの回数を大きく削減できることが確認できた。これに加え、実行可能ゴールのキュー管理のオーバヘッドがなくなったことにより、実行時間も減少し、従来方式ではスケジューリング・オーバヘッドの大きいもので2〜3倍、それほどオーバヘッドを生じないものでも、1.3倍程度の速度向上を得ることができた。また、並列実行の場合にも、逐次実行と同程度の速度向上を得ることができ、粒度向上によるアイドル時間増加の問題も、reply first 方式により解消できることが確認できた。

今後の課題として、スレッド化されたゴールの実行方式については、スタックの管理方法などに改良の余地がある。また、並列実行に関しては、より大規模なプログラムなどによる、詳細な性能評価が必要である。さらに、5.3.2 項で述べたスレッドの動的粒度制御についても、実装および評価を行う必要がある。

Chapter 6

結論

6.1 本論文のまとめ

本論文では、並列論理型言語 KL1 の静的解析手法と、これを利用した実行最適化手法について述べた。

2 章では、本論文の対象である並列論理型言語 KL1 と、本研究で実装の対象とした処理系 KLIC について、概略を説明した。KLIC をはじめとする従来の KL1 処理系では、動的オーバヘッドにより充分な実行性能を発揮できていない。これを最適化し、性能向上を図るのが本研究の目的であり、本章では主な問題点として、ゴール・スケジューリングおよび並列実行時のノード間通信におけるオーバヘッドを指摘した。

3 章では、KL1 を対象とした静的解析手法の提案を行った。KL1 処理系の動的オーバヘッドを削減するには、静的解析により事前にある程度の振る舞いを調べ、生成コードを最適化して動的処理を減らす方法が有効である。本章では、まず従来の静的解析手法を概観し、比較を行った。次に、これらを踏まえ、KL1 の性質と本研究の目的に適した静的解析手法を提案し、その内容を詳述した。本解析手法は制約充足による型解析を行うものであり、型情報を拡張することによって、様々な解析に応用できる。また、ゴールの実行順が動的に変化するなど、KL1 特有の解析上の問題を、制約充足を用いることにより解決している。

4 章では、この静的解析手法を利用し、ノード間通信の最適化を行った。本章では、まずノード間通信の問題点を述べた。従来の処理系では、要求駆動的なデータのやりとりにより細粒度の通信が発生し、そのオーバヘッドが速度低下につながる。次に、この問題の解決策として、必要なデータのみをまとめ送りすることで、無駄なデータ送信を生じずに通信回数を削減する、選択的一括送信手法を提案し、その概要を説明した。この手法では、まず静的解析により、各並列プロセスで参照されるデータ要素を調べる。次に、この情報より、必要な要素をまとめ送りする選択的一括送信コードを生成する。実行時には、データ要求に対し、対応する送信コードを実行することによって、必要なデータの選択的一括送信を実現する。本章では、この手法のために 3 章の解析手法を拡張した。また、選択的一括送信コードを生成し、実行時にこれを利用する方法について、それぞれ詳しく説明した。最後に、KLIC における本手法の実装について述べ、性能評価を行った。この結果、総転送量を増加させることなく、通信回数を 2/3〜1/200 程度に大きく削減することができた。また、多くの場合は 2〜5 倍程度の速度向上が得られ、本手法の有効性が示された。

5 章でも、3 章の静的解析手法を利用し、ゴール・スケジューリングの最適化を行った。本章

では、まずゴール・スケジューリングの問題点を述べた。従来の方式では、ゴールを並行制御の単位とするため細粒度の動的スケジューリングが行われ、そのオーバヘッドが大きなものになる。次に、この問題の解決策として、逐次化可能なゴール群を静的スケジューリングする、ゴールのスレッド化手法を提案し、その概要を説明した。この手法では、静的解析によりゴール間の依存を調べ、静的に実行順序を固定できるゴール群を1スレッドとすることで、スレッド内のゴールの動的スケジューリング・オーバヘッドを削減する。また、スレッドを動的スケジューリングすることで、プログラムが持つ本質的な並行性を保つ。さらに、スレッド化により並行性の粒度が上がり、並列性が低下する問題に対し、他ノードから要求された値を生成するスレッドを優先的にスケジューリングする、reply first 方式を提案した。本章では、この手法のために 3 章の解析手法を拡張した。また、依存情報を利用してプログラムをスレッド化し、reply first 方式を実現するコードを生成する手法を詳しく述べた。さらに、実行時にスレッド内ゴールを逐次実行し、reply first 方式によりスレッドを動的スケジューリングする方法についても説明した。最後に、KLIC における本手法の実装について述べ、性能評価を行った。この結果、本手法により動的スケジューリングのオーバヘッドを大きく減らすことができるのに加え、スレッド内ゴールをスタックで管理することにより、ガーベジコレクションの回数削減にも効果があることが示された。この結果、従来方式でオーバヘッドの大きいプログラムで 2〜3 倍、小さいものでも 1.3 倍程度の速度向上が得られた。また、reply first 方式を用いることにより、並列実行でも並列性低下の問題を改善でき、性能向上が得られた。

以上の結果より、本研究で提案した静的解析手法は、適切な拡張を加えることで、様々な静的情報を得る手段として活用できることが示された。また、これを利用した 2 種類の最適化手法は、性能評価によりいずれの場合も高い効果が得られ、KL1 処理系の実行最適化手法として有効であることが確認できた。

6.2 今後の研究課題

今後の研究課題としては、次のようなものが挙げられる。

静的解析手法の改良と応用

本解析手法は型や依存といったプログラムの定性的な解析しか行っていない。これは、データサイズなどの定量的な解析が、論理型言語では一般に困難なためである。しかし、従来の研究において、プロセスの粒度をデータの大きさに対するオーダーで静的評価し、実行時に実データの大きさを当てはめて粒度評価を行う手法 [41] などが提案されており、本研究でもこうした手法を取り入れることは可能と思われる。

本研究の最適化では、選択的一括送信手法でデータをまとめすぎることにより、かえって並列性を低下させるような結果が起こりうる。これに対し、定量解析を併用することによって、データを一定量ごとにまとめ送りするといった、より適切な最適化が可能になることが期待できる。

一方、ゴール・スケジューリング最適化手法では、逐次化可能なゴール群をすべて 1 スレッドとしてしまうため、並列性の低下という問題が生じる。本手法ではこの解決策として reply first スケジューリングを提案したが、プログラムによってはスレッドの本数が極端に少なくなり、このスケジューリング方式でも並列性を改善できない事態が起こりうる。本研究のスレッド化手法は、スレッド内の任意のゴールを開始ゴールとして動的に新スレッドを生成できるため、動的な粒度制御が可能である。しかし、効率的な粒度制御を行うためには、どのゴールを新スレッドと

するべきかの指針となる、解析情報が必要である。ここで、粒度評価により各スレッドの大きさを静的評価できれば、一定以上の粒度のスレッドは分割するといった手法が可能になる。

また、本研究では、提案した静的解析手法の応用としてノード間通信とゴール・スケジューリングの最適化を行ったが、本解析手法は他の最適化手法にも応用できると考えられる。たとえば、依存解析を応用して各型の具体化・参照ゴールの情報を得れば、不要になったデータ領域をその場で再利用するコンパイル時ガーベジコレクションに利用できる。こうした他の最適化手法についても、解析手法の拡張方法と合わせて、今後、検討を行っていく必要がある。

実装方式の改良

本研究で行った実装は、実現の容易さに比重を置いた方式をとっているため、実行効率にはまだ改良の余地がある。また、解析情報を利用し実行系でより効率的な動的制御を行うことで、さらに性能改善が期待できる。

選択的一括送信手法では、選択的一括送信コードにKL1を採用したため、送信処理自体のオーバヘッドがある程度、大きなものになっている。この部分をCで記述し直すことにより、実装の容易さや移植性と引き替えに、より高い性能が期待できる。

また、プログラムによっては、メッセージ粒度を上げる代わりにデータ生成側が要素の具体化ごとに細粒度のメッセージを送る、単方向送信が効果的と考えられる。これについては、今後、性能評価をもとに選択的一括送信との比較を行い、両方式の利害得失を検討することによって、最終的には解析結果を用いて両方式を使い分けるような実装が可能と思われる。

さらに、4.8節で触れたように、選択的一括送信の粒度制御と遅延を動的に行うことによって、性能を改善できるという結果が得られている[52]。この方式の現在の実装は、実行時のノード間通信における反応時間よりフィードバック制御を行うというものであり、一般の選択的一括送信に単純に適用した場合には、効果が得られない場合や制御のオーバヘッドが問題になる場合が起こりうる。そこで、解析情報を利用し、特定の条件を満たす場合のみ粒度制御を行ったり、粒度制御自体にも解析情報による最適化を加えることによって、さらなる性能改善が期待できる。

一方、ゴール・スケジューリング最適化手法では、現在の実装にはゴールスタックの管理方式に改良の余地がある。本研究のスレッド化では、再帰ループを含む非常に粒度の大きなスレッドが生成されうる。このため、あまりスタックを消費しないスレッドがある一方、再帰により大量のゴールがスタックを消費するスレッドが生じる。現在の実装では、ゴールスタックがあふれた場合にスタックのチェーンを形成するため、後者のようなスレッドでは余分なオーバヘッドを生じる。ここで、粒度解析の結果よりスタックの消費量を見積もることができれば、より効率的なスタックの割当が可能になる。

さらに、現在の方式では、スタック上にゴールレコードのみを保持しているが、変数や具体値もスタックにおくことができれば、さらにヒープの消費量を減らし、ガーベジコレクションのオーバヘッドを削減できる。実行が終われば即座に破棄できるゴールレコードと異なり、変数や具体値は複数のゴール間で共有される可能性があるため、一般にはスタック上に保持することができない。しかし、静的解析により、そのスタック上で自身の上に積まれるゴールのみが参照するといった条件を判定できれば、安全な場合に限ってスタック上に配置することが可能になる。

また、現在の実装は、5.3.2項で述べたスレッドの動的粒度制御を行っていない。これの単純な実装方法としては、ノード内のスレッド数を監視し、閾値を下回った場合には実行スレッド内のゴールの一部を新スレッドとして分割するという手法が考えられる。しかし、効果的な動的粒度制御を行うには、粒度解析により各ゴール下の仕事量を見積もり、分割したスレッドが常に一定

以上の粒度になるような工夫が必要である。

他の言語の実行最適化

本研究で提案・実装した解析手法および実行最適化手法は、KL1 の言語仕様や KLIC の設計に依存する部分があるため、そのままでは他の言語に適用することはできない。しかしながら、手法の基本的な部分は、並列論理型言語一般に応用可能である。したがって、型や組込述語、並行／並列制御の方法など、言語により仕様の異なる部分について本手法に修正を加えることで、他の言語処理系の改良にも利用できるものと考えられる。

また、論理型言語以外の最適化についても、言語上は明示されていない情報を静的解析により取得し、これを利用して実行を効率化するというアプローチは有用である。たとえばオブジェクト指向型言語では、オブジェクトのカプセル化や継承といった機構により安全性や部品の再利用性を高めた反面、実行時の型判定などのオーバヘッドを生じるようになっている。この種の問題についても、静的な型解析を行い、判定コードを最適化する手法が有効である。こういった分野についても、今後、研究を行っていく必要があると考える。

謝辞

本研究を行う機会を与えて下さり、本研究をまとめるに際し御指導・御鞭撻を賜った、京都大学工学部 富田 眞治教授に、深甚なる謝意を表します。また、本論文執筆に際して御指導を賜った、京都大学工学部 佐藤 雅彦教授、湯淺 太一教授に、深く感謝の意を表します。

本研究に関して適切な御指導・御助言を賜った、豊橋技術科学大学情報工学系 中島 浩教授、京都大学工学部 森 眞一郎助教授に、心より感謝の意を表します。とくに中島先生には、本研究を進めるにあたって、終始御指導頂きました。

五島 正裕助手を始めとする富田研究室の方々には、本研究に関して日頃、数々の御討論を頂きました。ここに深く感謝致します。その中でも、田沼 仁氏、伊川 雅彦氏、黒田 圭氏には、本研究での実装面で、多大な御尽力を頂きました。ここに感謝致します。

KL1 および KLIC に関する数々の技術的資料や御助言を提供して頂いた、(財) 新世代コンピュータ技術開発機構 (ICOT) (現在、先端情報技術研究所 (AITEC)) と、東京大学工学部 近山 隆教授、早稲田大学理工学部 上田 和紀教授を始めとする KLIC タスクグループの方々に、深く感謝致します。

(株) 富士通研究所には、並列計算機 AP1000 の実行環境を御提供頂き、またその利用にあたって数々の情報を頂きました。ここに感謝の意を表します。

シンガポール国立大学の Roland Yap 先生には、数多くの御助言を頂きました。ここに深くお礼申し上げます。

最後に、研究生活を支えてくれた、父の完、母の宣子、姉の幸子、妹の裕子に、心より感謝します。

著者発表論文

主要論文

[1] 窪田 昌史, 三吉 郁夫, 大野 和彦, 森 眞一郎, 中島 浩, and 富田 眞治. 不規則アクセスを伴うループの並列化コンパイル手法 – inspector/executor アルゴリズムの高速化 –. 情報処理学会論文誌, 35(4):532–541, April 1994.

[2] Atsushi Kubota, Ikuo Miyoshi, Kazuhiko Ohno, Shin ichiro Mori, Hiroshi Nakashima, and Shinji Tomita. A parallelizing compiler technique for loops with irregular accesses – new algorighms to improve the performance of the inspector/executor –. In Lubomir F. Bic, Alexandru Nicolau, and Mitsuhisa Sato, editors, *Parallel Language and Compiler Reseach in Japan*, chapter 13, pages 313–323. Kluwer Academic Publishers, 1995.

[3] 大野 和彦, 伊川 雅彦, 森 眞一郎, 中島 浩, and 富田 眞治. 静的解析による並列論理型言語 KL1 のメッセージ通信最適化. 情報処理学会論文誌, 38(8):1638–1648, August 1997.

[4] K. Ohno, M. Ikawa, M. Goshima, S. Mori, H. Nakashima, and S. Tomita. Improvement of message communication in concurrent logic language. In *Proceeding of the Second International Symposium on Parallel Symbolic Computation PASCO'97*, pages 156–164, 1997.

[5] K. Ohno, M. Ikawa, M. Goshima, S. Mori, H. Nakashima, and S. Tomita. Efficient goal scheduling in concurrent logic language using type-based dependency analysis. In *LNCS1345 Advances in Computing Science – ASIAN'97*, pages 268–282. Springer-Verlag, 1997.

その他の発表論文

[6] 大野 和彦, 森 眞一郎, 中島 浩, and 富田 眞治. 動的負荷分散のための並列木探索 (DTS) アルゴリズムの拡張 – 並列計算機 AP1000 でのライブラリ化 –. In 情処研報 *93-PRG-13*, pages 57–64, August 1993.

[7] 大野 和彦, 中島 浩, and 富田 眞治. 静的解析による並列論理型言語の実行最適化. In 情処研報 *94-PRG-18*, pages 17–24, July 1994.

[8] 大野 和彦, 伊川 雅彦, 森 眞一郎, 中島 浩, and 富田 眞治. 並列論理型言語 KL1 の最適化手法. In 情処研報 *94-OS-67*, pages 183–190, December 1994.

[9] 伊川 雅彦, 大野 和彦, 中島 浩, and 富田眞治. 並列論理型言語処理系 KLIC における通信の高速化. In 情処研報 *95-PRO-2*, pages 105–112, 1995.

[10] 伊川 雅彦, 大野 和彦, 五島 正裕, 森 眞一郎, 中島 浩, and 富田 眞治. KLIC におけるゴール・スケジューリング最適化. In 情処研報 *96-PRO-10*, pages 43–48, 1996.

参考文献

[11] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[12] M. Ishikawa, M. Hoshida, M. Hirosawa, T. Toya, K. Onizuka, and K. Nitta. Protein sequence analysis by parallel inference machine. In *Proc. of the Fifth Gener. Comp. Sys.*, pages 294–299, 1992.

[13] H. Fujita and R. Hasegawa. A model-generation prover in KL1 using ramified stack algorithm. In *Proc. of the Eighth Intl. Conf on Logic Programming*, 1991.

[14] T. Chikayama, T. Fujise, and D. Sekita. A portable and efficient implementation of KL1. In *Proc. 6th Intl. Symp. PLILP'94*, pages 25–39, 1994.

[15] 近山 隆. *KLIC* ユーザーズ　マニュアル, January 1995.

[16] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proc. 6th Intl. Conf. and Symp. on Logic Programming*, pages 436–451, 1989.

[17] T. Chikayama. Operating system PIMOS and kernel language KL1. In *FGCS'92*, pages 73–88, 1992.

[18] K. Hirata and R. Yamamoto. Parallel and distributed implementation of concurrent logic programming language KL1. In *FGCS'92*, pages 436–459, 1992.

[19] K. Ueda. Guarded horn clauses:a parallel logic programming language with the concept of a guard. Technical Report 208, ICOT, 1986.

[20] K. Ueda. Guarded horn clauses. In *LNCS221 Logic Programming '85*, pages 168–179. Springer-Verlag, 1986.

[21] ICOT PIMOS 開発グループ. *PIMOS*マニュアル *(第 3.0*版*)*—プログラミング編—, November 1991.

[22] T. Chikayama. *KLIC User's Manual.* ICOT, March 1995.

[23] 六沢 一昭, 仲瀬 明彦, 近山 隆, and 藤瀬 哲朗. KLIC 分散メモリ処理系の設計と初期評価. In 並列処理シンポジウム *JSPP'95*, pages 153–160, May 1995.

[24] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience 2*, pages 315–339, December 1990.

[25] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *Journal of Logic Programming*, 5:207–229, 1988.

[26] M. Bruynooghe and G. Janssens. An instance of abstract interpretation integrating type and mode inferencing. In *5th International Conference on Logic Programming*, pages 669–683, 1988.

[27] R.Sundararajan, A.V.S.Sastry, and E.Tick. Variable threadedness analysis for concurrent logic programs. In *Joint International Conference and Symposium on Logic Programming*, pages 493–508, November 1992.

[28] M. Codish, M. Falaschi, and K. Marriott. Suspention analysis for concurrent logic programs. In *International Conference on Logic Programming*, pages 331–345, June 1991.

[29] A. King and P. Soper. Schedule analysis of concurrent logic programs. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 478–492, November 1994.

[30] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[31] A. Mycroft and R. A. O'Keefe. A polymorphic type system for prolog. *Artificial Intelligence*, 23:295–307, 1984.

[32] J. Zobel. Derivation of polymorphic types for prolog programs. In *4th International Conference on Logic Programming*, pages 816–838, 1987.

[33] R. Warren and S. K. Debray. On the practicality of global flow analysis of logic programs. In *5th International Conference on Logic Programming*, pages 684–699, 1988.

[34] K. Muthukumar and M. Hermenegildo. Determination of variable dependence information through abstract interpretation. In *Proceedings of the North American Conference of Logic Programming*, pages 166–185, November 1992.

[35] P. Mishra and U. S. Reddy. Declaration-free type checking. In *12th ACM Symposium on Principle of Programming Language*, pages 7–21, January 1985.

[36] A. K. Bansal and L. Sterling. An abstract interpretation scheme for logic programs based on type expression. In *Proceedings of the International Conference on Fifth Generation Computer Systems'88*, pages 422–429, 1988.

[37] Z. Somogyi. A system of precise modes for logic programs. In *4th International Conference on Logic Programming*, pages 769–787, 1987.

[38] K. Ueda and M. Morita. Message-oriented parallel implementation of Moded Flat GHC. In *Proc. Intl. Conf. on FGCS'92*, pages 799–808, 1992.

[39] K. Ueda and M. Morita. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, November 1994.

[40] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21-3:412–510, September 1989.

[41] E.Tick. Compile-time granularity analysis for parallel logic programming languages. In *New Generation Computing*, pages 325–337, July 1990.

[42] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principle of Programming Language*, pages 238–252, January 1977.

[43] 伊川 雅彦. 並列論理型言語処理系 KLIC における通信の高速化. 京都大学特別研究報告書, February 1995.

[44] H. Ishihata, T. Horie, S. Inano, T. Shimizu, and S. Kato. An architecture of highly parallel computer AP1000. In *IEEE Pacific Rim Conf. on Communications, Computers and Signal processing*, pages 13–16, May 1991.

[45] 石畑宏明 and 堀江健志. MIMD アーキテクチャとそのアルゴリズム — 1. 並列アルゴリズムと並列アーキテクチャ理論と実際, pages 71–84, June 1992.

[46] 富士通株式会社. *AP1000* ライブラリマニュアル *[ホスト] C* 言語インタフェース.

[47] 富士通株式会社. *AP1000* ライブラリマニュアル *[セル] C* 言語インタフェース.

[48] 清水俊幸, 堀江健志, and 石畑宏明. 高速メッセージハンドリング機構 —AP1000 における実現—. 並列処理シンポジウム *JSPP'92*, pages 195–202, June 1992.

[49] 田沼 仁. 並列論理型言語 KL1 の AP1000 への実装. 京都大学特別研究報告書, February 1994.

[50] M. Fujita, R. Hasegawa, M. Koshimura, and H. Fujita. Model generation theorem provers on a parallel inference machine. In *Proc. of Fifth Gener. Comp. Sys.*, 1992.

[51] H. Nakashima and Y. Inamura. An efficient message transfer mechanism bypassing transit processors. In 並列処理シンポジウム *JSPP'92*, pages 123–130, June 1992.

[52] 黒田 圭. KL1 言語処理系 KLIC の動的粒度制御による通信高速化. 京都大学特別研究報告書, February 1997.

[53] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning - a new algorithm for partitioning non-strict programs into sequential threads. In *POPL'95*, pages 259–271, 1995.

[54] 荒木拓也 and 田中英彦. Commited-choice 型言語 Fleng における静的粒度最適化. In 情報処理学会研究報告 *96-PRO-8*, pages 109–114, August 1996.

[55] D. H. D. Warren. An abstract prolog instruction set. Technical Report 309, SRI International, 1983.

[56] 伊川 雅彦. 並列論理型言語 KL1 におけるゴール・スケジューリング 最適化手法. Master's thesis, 京都大学大学院, February 1997.

[57] B. C. Massey and E. Tick. Sequentialization of parallel logic programs with mode analysis. In *4th International Conference on Logic Programming and Automated Reasoning*, pages 205–216, 1993.

[58] M. Nilsson and H. Tanaka. Fleng prolog - the language which turns supercomputers into prolog machines. In *LNCS264*. Springer-Verlag, 1989.

[59] 日高 康雄, 小池 汎平, and 田中 英彦. Pie64 における複合粒度並列処理を用いた最適粒度制御 – 細粒度並列と粗粒度並列; 二つの異質な世界の分離と融合 –. In 並列処理シンポジウム *JSPP'95*, pages 115–122, 1995.